目录

[一、基础篇: 通过log分析KE 1](#_Toc448152917)

[1、KE构架 1](#_Toc448152918)

[①、KE概念 1](#_Toc448152919)

[②、KE类别 1](#_Toc448152920)

[③、常用调试方法 2](#_Toc448152921)

[2、kernel 空间布局 2](#_Toc448152922)

[①、ARM32bit kernel布局 2](#_Toc448152923)

[②、ARM64bit kernel布局 3](#_Toc448152924)

[3、了解printk 4](#_Toc448152925)

[①、AEE db 4](#_Toc448152926)

[4、ram console 4](#_Toc448152927)

[①、系统重启时关键信息 4](#_Toc448152928)

[5、前期异常处理 6](#_Toc448152929)

[①、软件异常捕获 6](#_Toc448152930)

[6、die()流程 7](#_Toc448152931)

[①、die()总流程 8](#_Toc448152932)

[②、\_\_die()流程 9](#_Toc448152933)

[③、异常类型信息 9](#_Toc448152934)

[④、CPU寄存器信息 9](#_Toc448152935)

[⑤、寄存器附近的内存 10](#_Toc448152936)

[⑥、调用栈 11](#_Toc448152937)

[⑦、PC附近指令 11](#_Toc448152938)

[7、panic()流程 12](#_Toc448152939)

[①、panic()流程 12](#_Toc448152940)

[②、panic通知链 12](#_Toc448152941)

[③、expdb 13](#_Toc448152942)

[8、nested panic 13](#_Toc448152943)

[二、进阶篇：ramdump分析 14](#_Toc448152944)

[9、ramdump文件种类 15](#_Toc448152945)

[①、为什么要用ramdump？ 15](#_Toc448152946)

[②、KE ramdump文件种类 15](#_Toc448152947)

[10、GUN tools 15](#_Toc448152948)

[11、AAPCS标准 15](#_Toc448152949)

[12、GDB调试 15](#_Toc448152950)

[13、crash调试 16](#_Toc448152951)

[①、什么是Crash？ 16](#_Toc448152952)

[②、安装/使用方法 16](#_Toc448152953)

[③、常用命令 16](#_Toc448152954)

[④、参考 17](#_Toc448152955)

[14、trace32调试 18](#_Toc448152956)

[三、进阶篇：在线分析 18](#_Toc448152957)

[15、JTAG调试 18](#_Toc448152958)

[16、KDB调试 18](#_Toc448152959)

[四、扩展篇：深入Linux内核 18](#_Toc448152960)

[17、kernel常用模块/结构 18](#_Toc448152961)

[五、实例篇：案例分析 18](#_Toc448152962)

[18、BUG at \_\_schedule\_bug 18](#_Toc448152963)

[19、BUG at \_\_get\_vm\_area\_node 20](#_Toc448152964)

[20、BUG at dmp\_wd\_handler 21](#_Toc448152965)

[21、BUG at dpm\_drv\_timeout 22](#_Toc448152966)

[22、BUG at check\_bytes\_and\_report 22](#_Toc448152967)

[23、BUG at \_i2c\_deal\_result 27](#_Toc448152968)

[24、BUG at mt\_pmic\_wrap\_irq 28](#_Toc448152969)

[25、调用kmalloc前关闭抢断引起KE 29](#_Toc448152970)

[26、timer重复初始化引起KE 32](#_Toc448152971)

[27、factory mode 关机 KE 35](#_Toc448152972)

[28、变量没有锁保护 42](#_Toc448152973)

[29、通话中频繁亮暗屏KE 44](#_Toc448152974)

[30、work没有初始化就是用引起KE 47](#_Toc448152975)

[31、work重复初始化引起KE 49](#_Toc448152976)

[32、tasklet没有初始化就是用引起KE 52](#_Toc448152977)

[33、通话中遮住psensor KE 55](#_Toc448152978)

[34、use after free引起KE 58](#_Toc448152979)

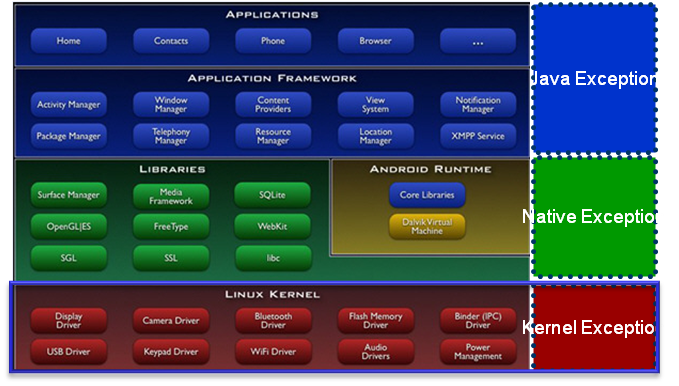
[35、内存泄露引起KE 62](#_Toc448152980)

一、基础篇: 通过log分析KE

1、KE构架

①、KE概念

Android OS由3层组成，最底层是kernel，上面是native bin/lib，最上层是java层：



任何软件都有可能发生异常，比如野指针，跑飞、死锁等等。

异常发生在kernel层，我们就叫它为KE（kernel exception），同理，发生在native就是NE，java层就是JE。这篇文章仅关注底层的KE。

②、KE类别

kernel有2中崩溃类别，

**oops** (**类似assert**，有机会恢复)

* oops是美国人比较常有的口语。就是有点意外，吃惊，或突然的意思。内核行为表现为通知感兴趣模块，**打印各种信息，如寄存器值，堆栈信息**…
* 当出现oops时，我们就可以根据寄存器等信息调试并解决问题。
* /proc/sys/kernel/panic\_on\_oops为1时导致panic。我们**默认设置为1**，即oops会发生panic。

**panic**

* Panic – 困惑，恐慌，它表示Linux kernel遇到了一个不知道该怎么继续的情况。内核行为表现为通知感兴趣模块，**死机或者重启**。
* 在kernel代码里，有些代码加了错误检查，发现错误可能直接调用了panic()，并输出信息提供调试。

其实不管分类几种，都表示kernel出现故障，需要修复。那如何调试呢？就要看在发生异常时留了哪些信息帮我们定位问题了。

③、常用调试方法

凡是程序就有bug。bug总是出现在预料之外的地方。据说世界上第一个bug是继电器式计算机中飞进一只蛾子，倒霉的飞蛾夹在继电器之间导致了计算机故障。由于这个小虫子，程序中的错误就被称为了bug。

有Bug就需要Debug，而调试是一种很个性化的工作，十个人可能有十种调试方法。但从手段上来讲，大致可分为两类，在线调试 (Online Debug) 和离线调试 (Offline Debug).

* 在线调试, Online debug, 指的是在程序的运行过程中监视程序的行为，分析是否符合预期。通常会借助一些工具，如GDB和Trace32等。有时候也会借助一些硬件设备的协助，如仿真器/JTAG，但是准备环境非常困难，而且用起来也很麻烦，除非一些runtime问题需要外很少使用。
* 离线调试, Offline debug, 指的是在程序的运行中收集需要的信息，在Bug发生后根据收集到的信息来分析的一种手段。通常也分为两种方式，一种是Logging，一种是Memory Dump。
  + Logging, 日志或者相关信息的收集，可以比较清晰的看到代码的执行过程，对于逻辑问题是一种有效的分析手段，由于其简单易操作，也是最为重要的一种分析手法。
  + Memory Dump, 翻译过来叫做内存转储，指的是在异常发生的时刻将内存信息全部转储到外部存储器，即将异常现场信息备份下来以供事后分析。是针对CPU执行异常的一种非常有效的分析手段。在Windows平台，程序异常发生之后可以选择启动调试器来马上调试。在Linux平台，程序发生异常之后会转储core dump，而此coredump可以用调试器GDB来进行调试。而内核的异常也可以进行类似的转储。

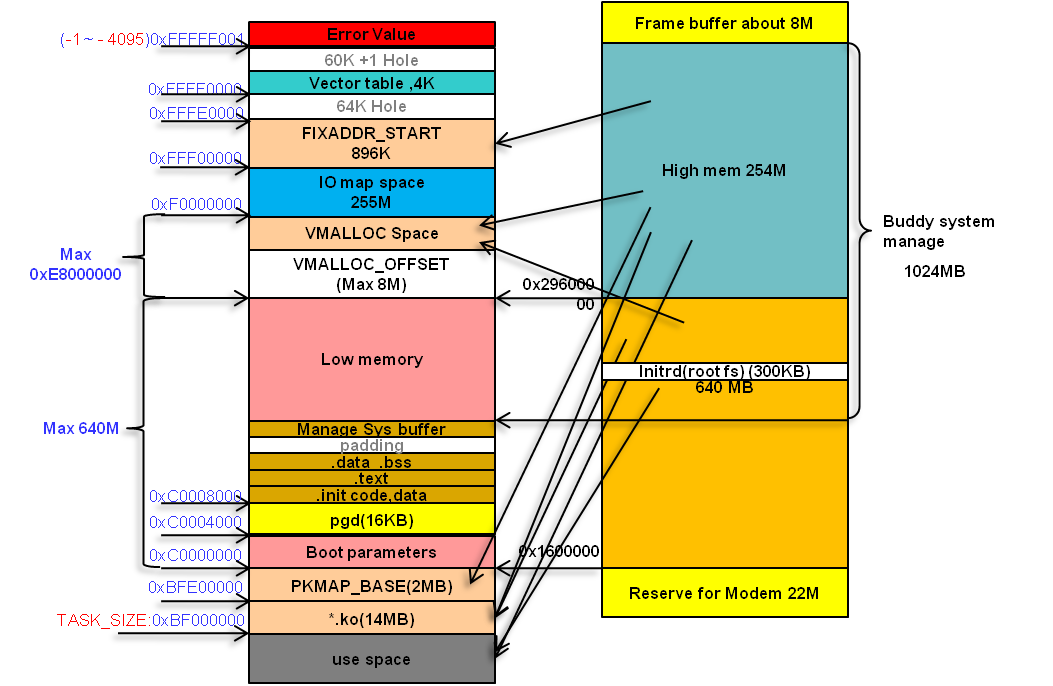
下面我们由浅入深剖析各种调试方法，先从logging开始吧。

****2、kernel 空间布局****

在分析KE前，你要了解kernel内存布局，才知道哪些地址用来做什么，可能会是什么问题。目前智能机已进入64bit，因此就存在32bit布局和64bit布局，下面一一讲解。

****①、ARM32bit kernel布局****

这是一张示意图（有些地址可能会有差异）



整个地址空间是4G，kernel被配置为1G，程序占3G。

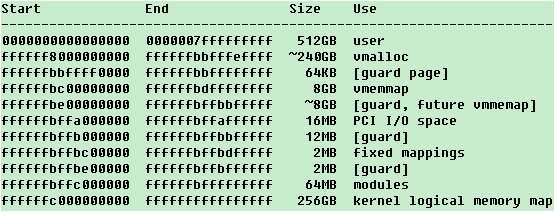
任何程序都有TEXT（可执行代码）,RW（数据段）,ZI段（未初始化数据段），kernel也有，对应的是.text,.data,.bss。而内核代码开始的地址是0xC0008000，前面放页表（起始地址为0xC0004000），如果支持模块（\*.ko）那么地址在0xBF000000。

由于kernel没办法将所有内存都映射进来，毕竟kernel自己只占1G，如果RAM超过1G，就无法全部映射。怎么办呢？只能先映射一部分了，这部分叫low memory。其他的就按需映射，VMALLOC区域就是用于按需映射的。

ARM的外设寄存器和内存一样，都统一地址编码，因此0xF0000000以上的一段空间用于映射外设寄存器，便于操作硬件模块。

0xFFFF0000是特殊地址，CPU用于存放异常向量表，kernel异常绝大部分都是CPU异常（MMU发出的abort/undef inst.等异常）。

****②、ARM64bit kernel布局****

****

这是39bit的kernel空间，由于多达512GB的空间，因此完全可以将整个RAM映射进来，0xFFFFFFC000000000之后就是一一映射了，就无所谓high memory了。

vmalloc还是存在，因为可以将不连续的物理内存拼接成连续的虚拟内存，可以解决部分内存碎片问题。而且外设寄存器也直接映射到vmalloc了，就没有32bit布局里的IO map space了。

modules对应的就是\*.ko内核模块了。

以上是粗略的说明，还需查看代码获取完整的分析信息（内核在不停演进，有些部分可能还会变化）。

3、了解printk

最初学编程时，大家一定用过printf()，在kernel里有对应的函数，叫printk()。

最简单的调试方法就是用printk()印出你想知道的信息了，而前面章节讲到oops/panic时，它们就通过printk()将寄存器信息/堆栈信息打印到kernel log buffer里。

了解kernel log对问题的调试将非常重要，这里有专门的课程介绍，请看：

* [MediaTek On-Line](https://onlinesso.mediatek.com/)> [Quick Start](https://onlinesso.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> Deep in MTK Turnkey Solution Logging Tools> Kernel log。

可以看到kernel log可以通过串口输出，也可以在发生oops/panic后将buffer保存成文件打包到db里，然后拿到串口log或db对kernel进行调试分析了。

通常手机会保留串口测试点，但要抓串口log一般都要拆机，比较麻烦。前面讲到可以将kernel log保存成文件打包在db里，db是什么东西？

①、AEE db

db是叫AEE（Android Exception Engine，集成在Mediatek手机软件里）的模块检查到异常并收集异常信息生成的文件，里面包含调试所需的log等关键信息。db有点像飞机的黑匣子。

对于KE来说，db里包含了如下文件（db可以通过GAT工具解开，请参考附录里的FAQ）：

|  |
| --- |
| \_\_exp\_main.txt：异常类型，调用栈等关键信息。  \_exp\_detail.txt：详细异常信息  SYS\_ANDROID\_LOG：android main log  SYS\_KERNEL\_LOG：kernel log  SYS\_LAST\_KMSG：上次重启前的kernel log  SYS\_MINI\_RDUMP：类似coredump，可以用gdb/trace32调试  SYS\_REBOOT\_REASON：重启时的硬件记录的信息。  SYS\_VERSION\_INFO：kernel版本，用于和vmlinux对比，只有匹配的vmlinux才能用于分析这个异常。  SYS\_WDT\_LOG：看门狗复位信息  ...... |

以上这些文件一般足以调试KE了，除非一些特别的问题需要其他信息，比如串口log等等。

4、ram console

    请参考：

* [MediaTek On-Line](https://onlinesso.mediatek.com/)> [Quick Start](https://onlinesso.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> Deep in MTK Turnkey Solution Logging Tools

①、系统重启时关键信息

    ram console除了保持last kmsg外，还有重要的系统信息，这些非常有助于我们调试。这些信息保存在ram console的头部ram\_console\_buffer里。

struct ram\_console\_buffer  
{  
 uint32\_t sig;  
 /\* for size comptible \*/  
 uint32\_t off\_pl;  
 uint32\_t off\_lpl; /\* last preloader: struct reboot\_reason\_pl\*/  
 uint32\_t sz\_pl;  
 uint32\_t off\_lk;  
 uint32\_t off\_llk; /\* last lk: struct reboot\_reason\_lk \*/  
 uint32\_t sz\_lk;  
 uint32\_t padding[3];  
 uint32\_t sz\_buffer;  
 uint32\_t off\_linux; /\* struct last\_reboot\_reason \*/  
 uint32\_t off\_console;  
  
 /\* console buffer\*/  
 uint32\_t log\_start;  
 uint32\_t log\_size;  
 uint32\_t sz\_console;  
};

这个结构体里的off\_linux指向了struct last\_reboot\_reason，里面保存了重要的信息：

struct last\_reboot\_reason  
{  
 uint32\_t fiq\_step;  
 uint32\_t exp\_type; /\* 0xaeedeadX: X=1 (HWT), X=2 (KE), X=3 (nested panic) \*/  
 uint32\_t reboot\_mode;  
  
 uint32\_t last\_irq\_enter[NR\_CPUS];  
 uint64\_t jiffies\_last\_irq\_enter[NR\_CPUS];  
  
 uint32\_t last\_irq\_exit[NR\_CPUS];  
 uint64\_t jiffies\_last\_irq\_exit[NR\_CPUS];  
  
 uint64\_t jiffies\_last\_sched[NR\_CPUS];  
 char last\_sched\_comm[NR\_CPUS][TASK\_COMM\_LEN];  
  
 uint8\_t hotplug\_data1[NR\_CPUS], uint8\_t hotplug\_data2;  
 uint64\_t hotplug\_data3;  
  
 uint32\_t mcdi\_wfi, mcdi\_r15, deepidle\_data, sodi\_data, spm\_suspend\_data;  
 uint64\_t cpu\_dormant[NR\_CPUS];  
 uint32\_t clk\_data[8], suspend\_debug\_flag;  
  
 uint8\_t cpu\_dvfs\_vproc\_big, cpu\_dvfs\_vproc\_little, cpu\_dvfs\_oppidx, cpu\_dvfs\_status;  
  
 uint8\_t gpu\_dvfs\_vgpu, gpu\_dvfs\_oppidx, gpu\_dvfs\_status;  
  
 uint64\_t ptp\_cpu\_big\_volt, ptp\_cpu\_little\_volt, ptp\_gpu\_volt, ptp\_temp;  
 uint8\_t ptp\_status;  
  
 uint8\_t thermal\_temp1, thermal\_temp2, thermal\_temp3, thermal\_temp4, thermal\_temp5;  
 uint8\_t thermal\_status;  
  
 void \*kparams;  
};

以上重要的信息在重启后将被打包到db里的**SYS\_REBOOT\_REASON**文件里。对这只文件的各个栏位解读请查看：

* [MediaTek On-Line](https://onlinesso.mediatek.com/)> [Quick Start](https://onlinesso.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 深入分析看门狗框架> 分析方法> HW reboot> HW reboot调试信息

5、前期异常处理

对于野指针、跑飞之类的异常会被MMU拦截并报告给CPU，这一系列都是硬件行为，具体请看：

* [MediaTek On-Line](https://onlinesso.mediatek.com/)> [Quick Start](https://onlinesso.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 深入分析Android native exception框架> 流程-异常处理
* 在上面章节里的内核异常处理流程，有一处不同，走到arm\_notify\_die()后，判断是kernel mode就直接调用die()了，而不是force\_sig\_info()

这类问题比较难定位，也是占KE比例的大头，原因通常是内存被踩坏、指针use atfer free等多种因素，在当时可能不会立即出现异常，而是到使用这块内存才有可能崩溃。

分析问题的手段也是多样化，比如用watch point，MMU protect或加debug code等（请参考附录FAQ）

①、软件异常捕获

在kernel代码里，一般会通过BUG(),BUG\_ON(),panic()来拦截超出预期的行为，这是软件主动回报异常的功能。

这些问题分析通常有固定的套路，请参考后面的：《实例篇: 案例分析》

**BUG()/BUG\_ON()实现**

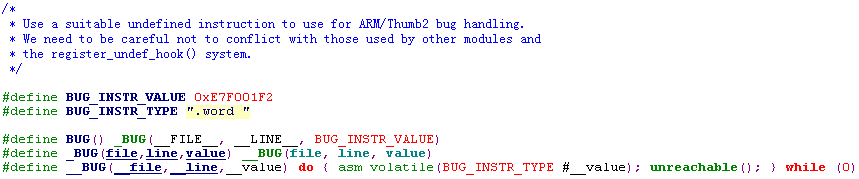
在内核调用可以用来方便标记bug，提供断言并输出信息。最常用的两个是BUG()和BUG\_ON()。当被调用的时候，它们会引发oops，导致栈的回溯和错误信息的打印。使用方式如下

if (condition)  
   BUG();  
或者 ：  
BUG\_ON(condition); //只是在BUG基础上多层封存而已：

 #define BUG\_ON(condition) do { if (unlikely(condition)) BUG(); } while(0)

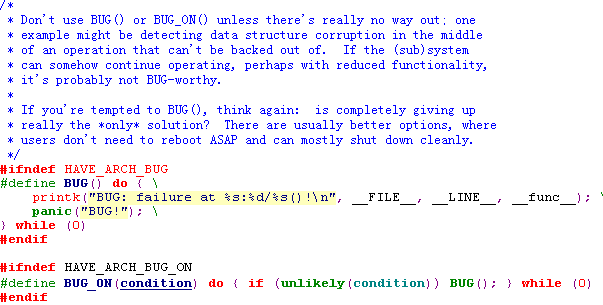
32bit kernel：

BUG() 的实现采用了埋入未定义指令（**0xE7F001F2**，记住这个值，log里看到这个值，你就应该知道是调用了BUG()/BUG\_ON()了）的方式

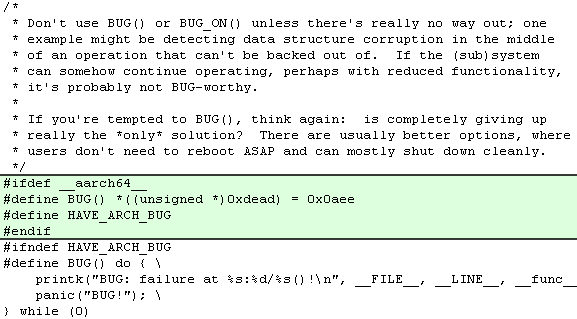


64bit kernel：

原生的kernel，BUG()是直接调用panic()的：



不过Mediatek修改了BUG()的实现，这样有更多的调试信息输出（die()有寄存器等信息输出）

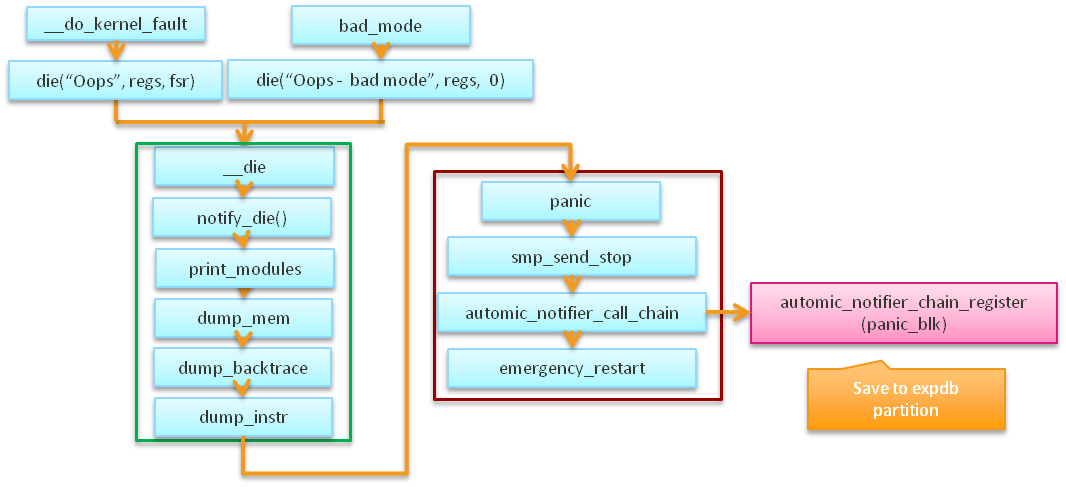


当你看到如下log时，就应该知道是BUG()/BUG\_ON()引起的了！

[  147.234926]<0>-(0)[122:kworker/u8:3]Unable to handle kernel paging request at virtual address 0000dead

6、die()流程

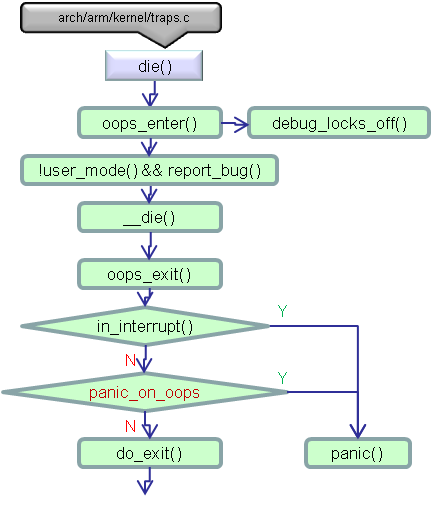
经过前面的流程，走到了die()函数，该函数主要输出便于调试的寄存器信息/堆栈信息等重要资料，我们通过log分析KE就是分析这些资料，因此要知道整个流程。die() => panic()的大致流程如下：



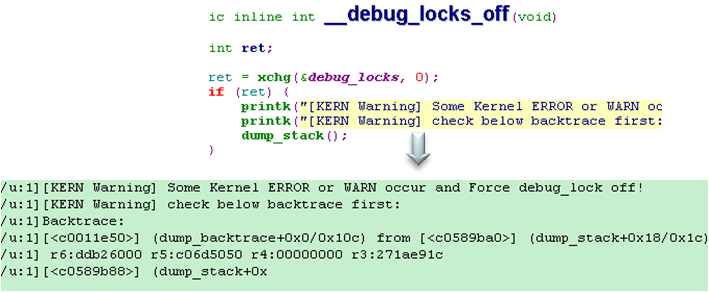
在学习这些流程时，建议结合代码和KE的log一起看，你就知道log里那些信息在代码哪处打印出来的了。

①、die()总流程

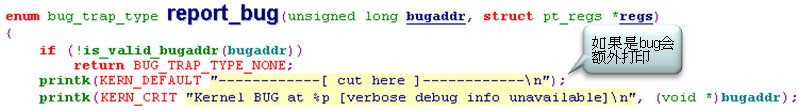
    先从die()入手，看下die()总流程：



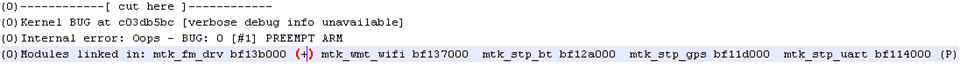
    走到debug\_locks\_off()就有log输出了，如下：



    如果这个异常是代码里调用BUG()/BUG\_ON()引起，那么有额外log说明：

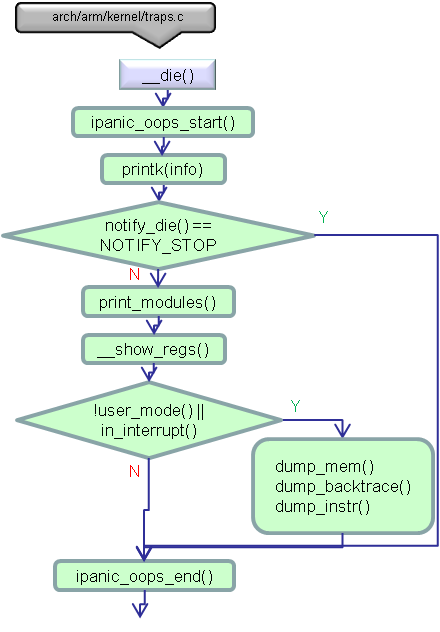


输出的log大致如下：



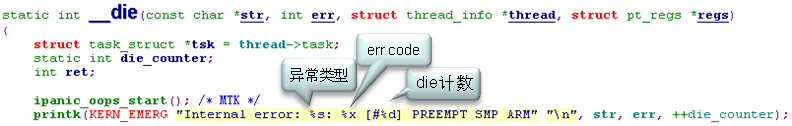
②、\_\_die()流程

    绝大部分的关键信息是由\_\_die()函数输出的，流程如下：

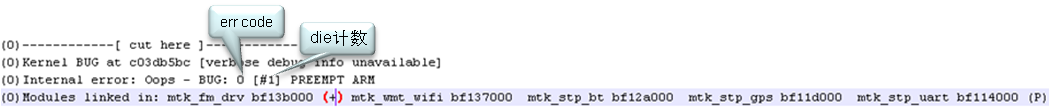


③、异常类型信息

开始印出异常类型等信息，看一份kernel log有没有oops，直接搜索关键字Internal error就可以了：

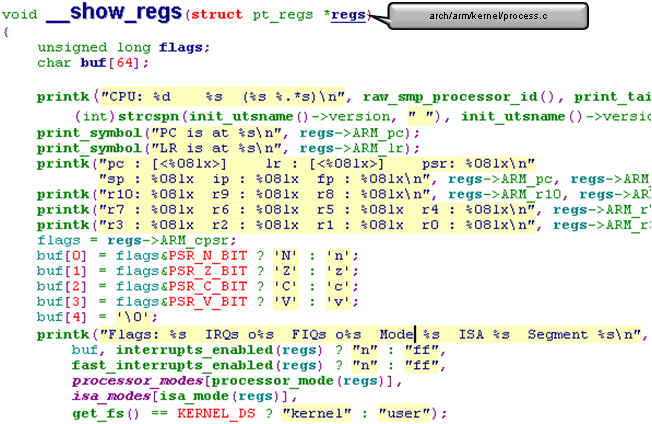


输出的信息大致如下：



④、CPU寄存器信息

然后是重要的CPU寄存器信息（32bit的代码，64bit类同）：

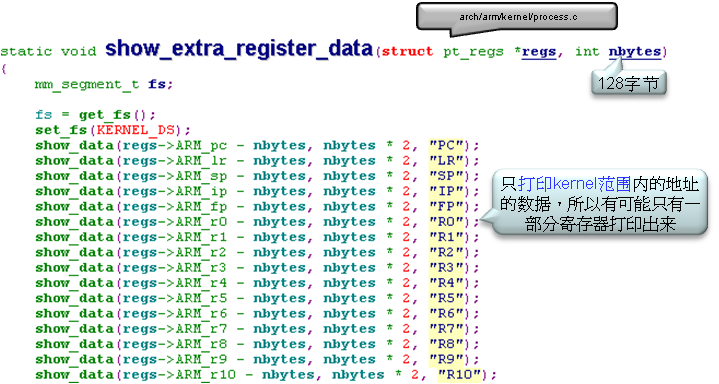


输出的信息大致如下：

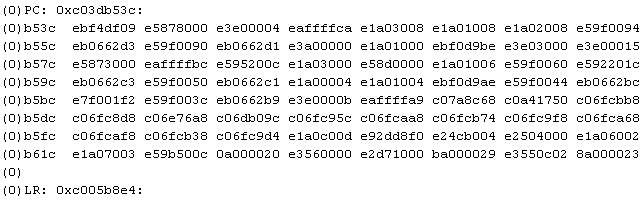


⑤、寄存器附近的内存

有助于我们分析问题的内存信息，问题很可能就出在里面。

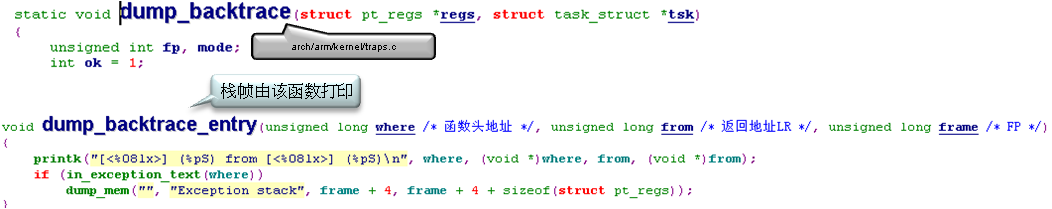


输出的信息大致如下：



⑥、调用栈

有时问题可以直接从调用栈看出来，由此可见调用栈是多么重要。

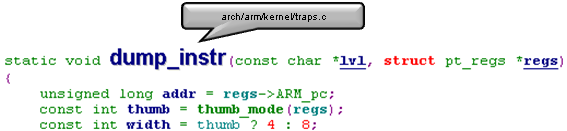


输出的信息大致如下：



⑦、PC附近指令

可以看到PC附近的指令：



输出的信息大致如下：



**分析log**

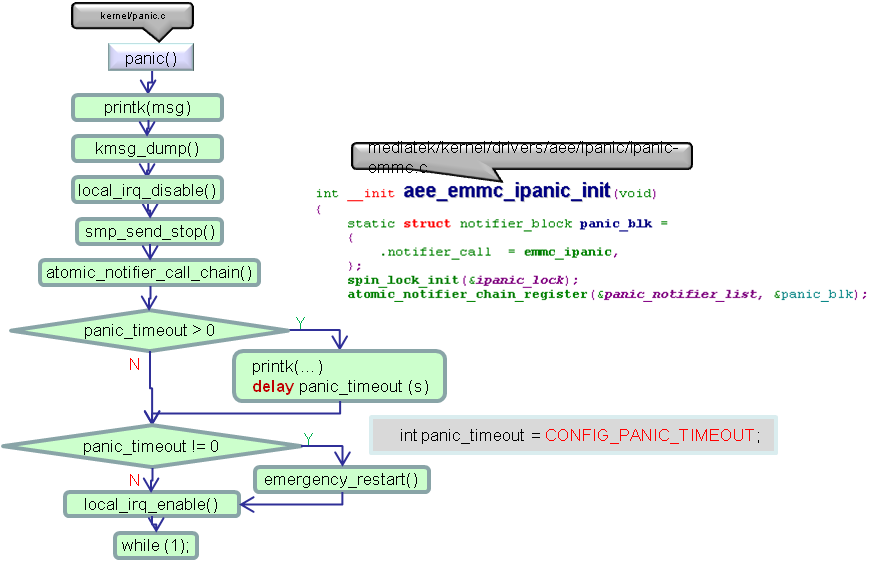
    到这里die()函数就完成了它的使命，将重要信息输出来了。接下来你要如何调试呢？这个就看个人的功力了，你可以：

* 通过PC指向的函数，用addr2line（后面的GNU tools有介绍）定位到哪只文件的哪一行，大致可以知道发生了什么，如果无法一下子定位，也可以通过结合printk()多次观察KE时的log排查。如果是由BUG()/BUG\_ON()引起的KE，则就可以着手修复问题了。
* 查看调用栈，有些时候调用栈可以说明流程，看看代码是否有按预期跑，如果没有，可以结合printk()定位问题。
* 如果你想看函数参数或全局变量信息，那么你需要用《进阶篇: ramdump分析》的知识调试了。

7、panic()流程

流程走到panic()就里死（异常重启）不远了，关键的信息已输出到kernel log。那么panic()做了什么呢？

****①、panic()流程****



panic()有标志性的log输出，大致如下：

panic1.png

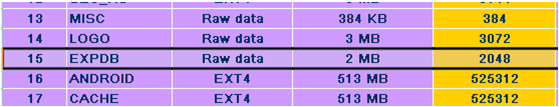
因此我们也可以通过搜索关键字Kernel panic查找是否有panic发生。

****②、panic通知链****

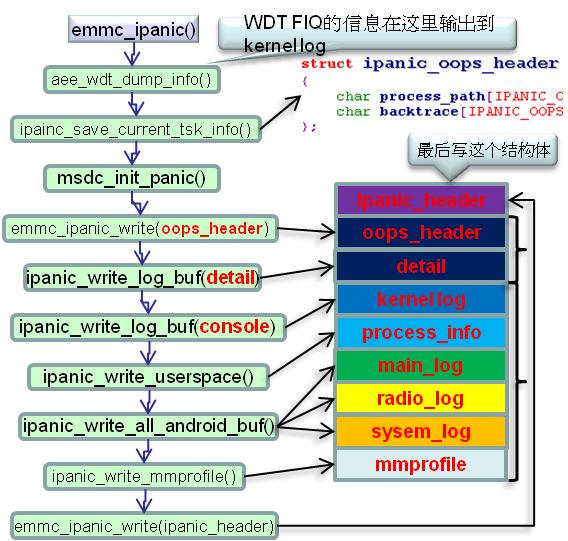
    panic()会调用栈通知链上的回调函数同时感兴趣的模块，比如我们的aee注册了回调函数，用于保存kernel log/mini dump等关键信息，并将其保存到emmc的expdb分区，等等重启后将其回读并保存成KE db。

****③、expdb****

重启过程DRAM会丢失，因此信息只能保存在flash上了，在分区表里有一项就是expdb了：



流程大致如下（版本不停演进，可能有很大变化，仅供参考）：



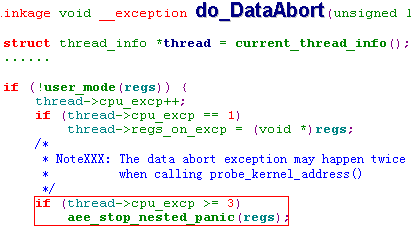
重启后，aee将回读aeedb分区资料并转化为KE db。

8、nested panic

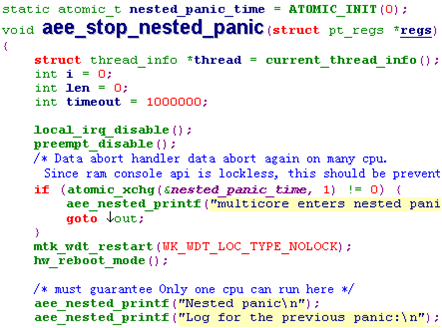
有时die()/panic()流程不一定能正常走完，可能走到某一步又发生了异常，则就形成了嵌套，这种情况，我们一般不会关注后面的异常，而是关注最开始的那个异常。

    为了避免异常嵌套，在发生第2次异常时，我们就拦截下来，我们在3个地方用于拦截nested panic：

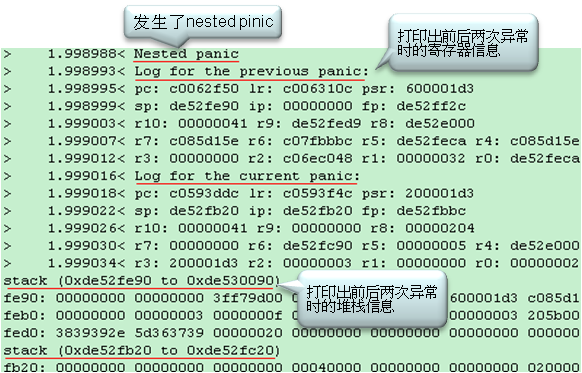
* do\_PrefetchAbort()
* do\_DataAbort()
* do\_undefinstr()



拦截后不走die()/panic()流程，因为这些流程可能会再次发生异常，走我们写的函数aee\_stop\_nested\_panic()函数：



在里面尽量少用kernel模块，很有可能也会发生异常，仅仅将寄存器等重要信息输出到ram console就等死（死循环等等看门狗复位！）。这时你抓回来的db里的SYS\_LAST\_KMSG就可以看到这些资料，大致如下（不同版本稍有区别）：



里面包含了寄存器信息、堆栈信息和调用栈，我们就可以通过工具（addr2line）还原当时异常的位置。

不过nested panic能参考的信息很少，不像普通的KE那样丰富。

二、进阶篇：ramdump分析

9、****ramdump文件种类****

①、为什么要用ramdump？

虽然debug方法有多种，但是都有其局限性。拿Logging来讲，Logging应该算是轻量级的Debug工具，并且在linux kernel默认就是打开的，在内核出现异常的时候我们可以很方便的得到下面这些信息：

* 出现问题前的log, log时间的长度取决于编译内核时配置的log buffer的大小。
* 问题发生时的call stack, 即当时的函数调用层次关系。

依靠这些信息，在大多数情况下我们都可以知道前面发生了什么，问题发生的大概原因是什么。但这里看到的也许并不是最根本的原因，因为它对某些情况的分析力度是有限的，比如:

* memory corruption, 当有内存区域被踩坏的时候logging的信息通常是不够的，此类情况我们大多时候会多复现几次来从中找到规律。而复现问题本身对于低概率的问题是一个灾难。
* 硬件不稳定导致的问题。许多时候某个硬件器件不稳定导致的系统性问题都是难以通过硬件手段来定位的，而通过logging下来的信息也仅仅能看到一部分。尤其是硬件不稳定导致的bit error更是不易发现的小角落。
* Log丢失的情况。在某些时候系统出现问题已经无法logging下来最后一段时间的系统行为，这也是log无能为力的地方。当然完善log机制可以得到改善，但是永远都无法在logging机制出现问题之前根除。

而这些Logging机制无法cover的问题，Memory dump作为离线调试的另外一个手段则正好可以发挥其威力。

②、KE ramdump文件种类

coredump是native的概念，我们将它扩展到kernel层，在db里面有**2只文件**可供我们调试：

* **SYS\_MINI\_RDUMP**: 该文件保存了发生异常的CPU寄存器信息及每个寄存器附近一段内存资料。包含的资料不多，所以文件也比较小，一般都是2M以内，这些基本足以用来调试。当然如果想看的资料超出该文件保存的内容时，就需要另外的方法了。
  + db里基本上都有这只文件（KK版本开始支持）。
* **SYS\_COREDUMP**: 该文件基本上将DRAM上的资料都保持下来了。里面有所有kernel管理的内存（排除了security内存/frame buffer），因此很大，比如3G DRAM的SYS\_COREDUMP的大小接近3G。
  + 有些项目默认是关闭的，详情请参考ramdump相关的FAQ。

有了dump文件，要怎么调试呢？需要借助debug工具，以下表格列出了工具可以支持的文件。

|  |  |  |
| --- | --- | --- |
| **工具** | **SYS\_MINI\_RDUMP** | **SYS\_COREDUMP** |
| gdb | 支持 | 部分支持（看不到task等内核信息） |
| crash | 不支持 | 支持 |
| trace32 | 支持 | 支持 |

下面我们一一讲解这些这些工具的使用方法。

10、GUN tools

* [MediaTek On-Line](https://onlinesso.mediatek.com/)> [Quick Start](https://onlinesso.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 深入分析Android native exception框架> 进阶篇: coredump分析> GNU tools

11、AAPCS标准

* [MediaTek On-Line](https://onlinesso.mediatek.com/)> [Quick Start](https://onlinesso.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 深入分析Android native exception框架> 进阶篇: coredump分析> AAPCS标准

12、GDB调试

* [MediaTek On-Line](https://onlinesso.mediatek.com/)> [Quick Start](https://onlinesso.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 深入分析Android native exception框架> 进阶篇: coredump分析> GDB调试

**①、使用方法**

    请查看：【FAQ13941】如何分析kernel panic？

用gdb调试SYS\_COREDUMP，是无法看到kernel信息，以及低于0xC0000000（32位）/ 0xFFFFFFC000000000（64位）的内存的。

13、crash调试

①、什么是Crash？

   当linux系统内核发生崩溃的时候，可以通过KEXEC+KDUMP等方式收集内核崩溃之前的内存，生成一个转储文件vmcore。内核开发者通过分析该vmcore文件就可以诊断出内核崩溃的原因，从而进行操作系统的代码改进。那么Crash就是一个被广泛使用的内核崩溃转储文件分析工具。

    前面讲过gdb调试方法，但gdb始终是调试native的工具，不支持kernel信息显示，比如task信息之类的。crash补足了这个短板，由Dave Anderson开发和维护的一个内存转储分析工具，是基于GDB开发的 (GDB适用于用户进程的coredump，而Crash扩展了GDB，使其适用于linux kernel coredump)，目前它的最新版本是7.0.5。在没有统一标准的内存转储文件的格式的情况下，Crash工具支持众多的内存转储文件格式，包括：

* Live linux系统
* kdump产生的正常的和压缩的内存转储文件
* 由makedumpfile命令生成的压缩的内存转储文件
* 由Netdump生成的内存转储文件
* 由Diskdump生成的内存转储文件
* 由Kdump生成的Xen的内存转储文件
* IBM的390/390x的内存转储文件
* LKCD生成的内存转储文件
* Mcore生成的内存转储文件

而我们前面讲到的SYS\_COREDUMP，则可以用crash来调试。

②、安装/使用方法

    请参考：【FAQ13939】搭建crash分析kernel ramdump平台

③、常用命令

crash使用gdb作为它的内部引擎，crash中的很多命令和语法都与gdb相同。如果曾经使用过gdb，就会发现crash并不是很陌生。如果想获得crash更多的命令和相关命令的详细说明，可以使用crash的内部命令help来获取：

|  |  |  |
| --- | --- | --- |
| **命令** | **说明** | **例子** |
| \* | 指针的快捷方式，用于代替struct/union | \*page 0xc02943c0：显示0xc02943c0地址的page结构体 |
| files | 显示已打开的所有文件的信息 | files 462：显示进程462的已打开文件信息 |
| mach | 显示与机器相关的参数信息 | mach：显示CPU型号，核数，内存大小等 |
| sys | 显示特殊系统的数据 | sys config：显示CONFIG\_xxx配置宏状态 |
| timer | 无参数。按时间的先后顺序显示定时器队列的数据 | timer：显示详细信息 |
| mod | 显示已加载module的详细信息 | mod：列出所有已加载module信息 |
| runq | 显示runqueue信息 | runq：显示所有runqueue里的task |
| tree | 显示基数树/红黑树结构 | tree -t rbtree -o vmap\_area.rb\_node vmap\_area\_root：显示所有红黑树vmap\_area.rb\_node节点地址 |
| fuser | 显示哪些task使用了指定的文件/socket | fuser /usr/lib/libkfm.so.2.0.0：显示使用了该文件的所有进程 |
| mount | 显示已挂载的文件系统信息 | mount：当前已挂载的文件系统信息 |
| ipcs | 显示System V IPC信息 | ipcs：显示系统中System V IPC信息 |
| ps | 显示进程状态 | ps：类似ps命令 |
| struct | 显示结构体的具体内容 | struct vm\_area\_struct c1e44f10：显示c1e44f10结构 |
| union | 显示联合体的具体内容，用法与struct一致 | union bdflush\_param：显示bdflush\_param结构 |
| waitq | 列出在等待队列中的所有task。参数可以指定队列的名称、内存地址等 | waitq buffer\_wait：显示buffer\_wait等待队列信息 |
| irq | 显示中断编号的所有信息 | irq 18：显示中断18的信息 |
| list | 显示链表的内容 | list task\_struct.p\_pptr c169a000：显示c169a000地址所指task里p\_pptr链表 |
| log | 显示内核的日志，以时间的先后顺序排列 | log -m：显示kernel log |
| dev | 显示数据关联着的块设备分配，包括端口使用、内存使用及PCI设备数据 | dev：显示字符/块设备相关信息 |
| sig | 显示一个或者多个task的signal-handling数据 | sig 8970：显示进程8970的信号处理相关信息 |
| task | 显示指定内容或者进程的task\_struct的内容 | task -x：显示当前进程task\_struct等内容 |
| swap | 无参数。显示已配置好的交换设备信息 | swap：交换设备信息 |
| search | 在给定范围的用户、内核虚拟内存或者物理内存搜索值 | search -u deadbeef：在用户内存搜索0xdeadbeef |
| bt | 显示调用栈信息 | bt：显示当前调用栈 |
| net | 显示各种网络相关的数据 | net：显示网络设备列表 |
| vm | 显示task的基本虚拟内存信息 | vm：类似于/proc/self/maps |
| btop | 把一个16进制地址转换成它的分页号 | N/A |
| ptob | 该命令与btop相反，是把一个分页号转换成地址 | N/A |
| vtop | 显示用户或内核虚拟内存所对应的物理内存 | N/A |
| ptov | 该命令与vtop相反。把物理内存转换成虚拟内存 | N/A |
| pte | 16进制页表项转换为物理页地址和页的位设置 | N/A |
| alias | 显示或建立一个命令的别名 | alias kp kmem -p：以后用kp命令相当于kmem -p |
| foreach | 用指定的命令枚举 | foreach bt：显示所有进程的调用栈 |
| repeat | 循环执行指定命令 | repeat -1 p jiffies：每个1s执行p jiffies |
| ascii | 把16进制表示的字符串转化成ascii表示的字符串 | ascii 62696c2f7273752f：结果为/usr/lib |
| set | 设置要显示的内容，内容一般以进程为单位，也可以设置当前crash的内部变量 | set -p：切换到崩溃进程的上下文环境 |
| p | print的缩写，打印表达式的值。表达式可以为变量，也可以为结构体 | N/A |
| dis | disassemble的缩写。把一个命令或者函数分解成汇编代码 | dis sys\_signal：反汇编sys\_signal函数 |
| whatis | 搜索数据或者类型的信息 | whatis linux\_binfmt：显示linux\_binfmt结构体 |
| eval | 计算表达式的值，及把计算结果或者值显示为16、10、8和2进制 | N/A |
| kmem | 显示当前kernel使用内存状况 | kmem -i：显示kernel使用内存状况 |
| sym | 显示符号所在的虚拟地址，或虚拟地址对应的符号 | sym jiffies：显示jiffies地址 |
| rd | 显示指定内存的内容。缺少的输出格式是十六进制输出 | rd -a linux\_banner：显示linux\_banner内容 |
| wr | 根据参数指定的写内存。在定位系统出错的地方时，一般不使用该命令 | wr my\_debug\_flag 1：修改my\_debug\_flag值为1 |
| gdb | 执行GDB原生命令 | gdb help：执行gdb的help命令 |
| extend | 动态装载或卸载crash额外的动态链接库 | N/A |
| q | 退出 | N/A |
| exit | 同q，退出 | N/A |
| help | 帮助命令 | N/A |

④、参考

Crash工具主页：<http://people.redhat.com/anderson/>

14、trace32调试

* [MediaTek On-Line](https://onlinesso.mediatek.com/)> [Quick Start](https://onlinesso.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> Trace32使用教程
* 《Trace32使用教程》里有KE分析章节，里面有讲解KE cmm脚本。

三、进阶篇：在线分析

15、JTAG调试

不是所有问题用前面的调试手段都能解决，比如故障现场时外设寄存器信息，动态信息等是离线分析无法做到的。这时jtag就派上用场了，虽然用起来麻烦，但在项目初期还是比较实用的。

jtag搭建/使用请到DCC上搜索查看文档：

* 《JTAG Debug User Guide.pptx》
* 《Debugger\_User Guide\_v5.1.docx》（版本可能会变化，比如5.2等）里的Hardware Assistant Debug Tool (ICE)章节

16、KDB调试

 JTAG必须要借助debug设备才行，用起来麻烦，有没有不用设备的在线调试方式呢？有！KDB就是一种，已是kernel的部分了。

KDB搭建/使用请到DCC上搜索查看文档：

* 《Debugger\_User Guide\_v5.1.docx》（版本可能会变化，比如5.2等）里的KDB章节

四、扩展篇：深入Linux内核

17、kernel常用模块/结构

到这里，基本上对KE调试有基本的了解，剩下的就是对kernel的熟悉程度了。越熟悉，调试起来越容易，也可以根据问题对症下药。

kernel内容非常庞大，可能不知道如何下手，建议先看Unix/Linux内核相关的书籍，了解内核的经典实现方法，然后再结合源码去研究Linux内核。这样做的原因是避免从一开始就陷入细节。

内核重点关注这几个部分：进程管理及调度，内存管理，文件及文件系统，Cache，I/O，SMP（多CPU）。

参考的书籍有（最好是看英文原版）：

* 《Linux内核设计与实现》
* 《Linux内核源代码情景分析》
* 《深入理解Linux内核》

等等。

另外要注意，linux kernel发展很快，有些模块/结构可能被移除或没有使用了，基本就不用关注了。

五、实例篇：案例分析

18、BUG at \_\_schedule\_bug

**异常现场：**

当你在SYS\_KERNEL\_LOG里看到如下log，那么就属于BUG at \_\_schedule\_bug一类问题了

|  |
| --- |
| [14005.496163]<5>-(5)[1020:AudioOut\_2]**BUG: scheduling while atomic:** AudioOut\_2/1020/0x00000002 [14005.496257]<5>-(5)[1020:AudioOut\_2]CPU: 5 PID: 1020 Comm: AudioOut\_2 Tainted: G W 3.10.72+ #2 [14005.496265]<5>-(5)[1020:AudioOut\_2]Call trace: [14005.496279]<5>-(5)[1020:AudioOut\_2][<ffffffc000088d2c>] dump\_backtrace+0x0/0x16c [14005.496291]<5>-(5)[1020:AudioOut\_2][<ffffffc000088ea8>] show\_stack+0x10/0x1c [14005.496303]<5>-(5)[1020:AudioOut\_2][<ffffffc000a7d56c>] dump\_stack+0x1c/0x28 [14005.496314]<5>-(5)[1020:AudioOut\_2][<ffffffc0000cd5c8>] \_\_schedule\_bug+0x58/0x84 [14005.496327]<5>-(5)[1020:AudioOut\_2][<ffffffc000a8db60>] \_\_schedule+0x6ac/0x858 [14005.496339]<5>-(5)[1020:AudioOut\_2][<ffffffc000a8dd30>] schedule+0x24/0x68 [14005.496351]<5>-(5)[1020:AudioOut\_2][<ffffffc000a8b15c>] schedule\_timeout+0x134/0x218 [14005.496364]<5>-(5)[1020:AudioOut\_2][<ffffffc0000ac8c0>] msleep+0x2c/0x40 [14005.496378]<5>-(5)[1020:AudioOut\_2][<ffffffc0008bf224>] RemoveMemifSubStream+0x4c/0x1c8 [14005.496390]<5>-(5)[1020:AudioOut\_2][<ffffffc0008c9708>] mtk\_pcm\_I2S0dl1\_close+0xb4/0xf8 [14005.496404]<5>-(5)[1020:AudioOut\_2][<ffffffc0008b1b68>] soc\_pcm\_close+0x118/0x1d4 [14005.496418]<5>-(5)[1020:AudioOut\_2][<ffffffc0008667bc>] snd\_pcm\_release\_substream.part.16+0x3c/0x90 [14005.496430]<5>-(5)[1020:AudioOut\_2][<ffffffc0008668a8>] snd\_pcm\_release+0x98/0xc8 [14005.496444]<5>-(5)[1020:AudioOut\_2][<ffffffc0001b2210>] \_\_fput+0x98/0x210 [14005.496455]<5>-(5)[1020:AudioOut\_2][<ffffffc0001b2434>] \_\_\_\_fput+0x8/0x14 [14005.496467]<5>-(5)[1020:AudioOut\_2][<ffffffc0000bdd88>] task\_work\_run+0x8c/0xe4 [14005.496478]<5>-(5)[1020:AudioOut\_2][<ffffffc0000888c8>] do\_notify\_resume+0x50/0x64 |

**代码位置：**

kernel-3.10/kernel/sched/core.c

static noinline void \_\_schedule\_bug(struct task\_struct \*prev)  
{  
 ......  
 printk(KERN\_ERR "BUG: scheduling while atomic: %s/%d/0x%08x\n",  
 prev->comm, prev->pid, preempt\_count());  
 ......  
 dump\_stack();  
 add\_taint(TAINT\_WARN, LOCKDEP\_STILL\_OK);  
 BUG\_ON(1); /\* 这里发生KE \*/  
}

**问题解读：**

scheduling while atomic的意思是：在原子上下文(关闭抢断)发起调度。

什么是原子上下文呢？不能被中断的上下文为原子上下文，比如：

* 已调用了preempt\_disable()（关闭抢断）后，就不能被调度了，一般持有spin lock会关闭抢断。
* 在中断上下文里。linux内核要求在中断处理的时候，不允许系统调度，不允许抢占，要等到中断处理完成才能做其他事情。
* 在软中断上下文里。

所以在原子上下文当然无法发起调度了。在持有spin lock或在中断里，都是**要尽快做完后离开**。

如果长时间处于原子上下文，就容易出问题，引起系统响应缓慢、卡顿，甚至引起看门狗复位或死锁。

**问题解决：**

首先要看为什么发起调度了，通常在中断处理函数中/拿到spin lock之后**调用了可能引起休眠的函数**，如semaphore,mutex,sleep之类的可休眠的函数。

检查下代码逻辑，调整这些代码移除在原子上下文外面。

比如前面的例子，在RemoveMemifSubStream()函数里调用的spin\_lock\_irqsave()拿到了spin lock，后面又调用了msleep()，这就引起了这个问题。解决方法是，delay还是要，所以将msleep()换成了udelay()就可以了。

**案例分析：**调用kmalloc前关闭抢断引起KE

19、BUG at \_\_get\_vm\_area\_node

**异常现场：**

当你在SYS\_KERNEL\_LOG里看到如下log，那么就属于BUG at \_\_get\_vm\_area\_node一类问题了

|  |
| --- |
| [ 271.881362]<0>-0)Internal error: Oops: 96000045 [#1] PREEMPT SMP ...... [ 280.012666]<0>-0)**PC is at \_\_get\_vm\_area\_node**.isra.28+0x1d4/0x1ec [ 280.012672]<0>-0)LR is at get\_vm\_area\_caller+0x2c/0x34 ...... [ 280.014883]<0>-0)Call trace: [ 280.014891]<0>-0)[<ffffffc00017df54>] **\_\_get\_vm\_area\_node**.isra.28+0x1d4/0x1ec [ 280.014898]<0>-0)[<ffffffc00017ea48>] get\_vm\_area\_caller+0x28/0x34 [ 280.014909]<0>-0)[<ffffffc0000948c4>] \_\_ioremap+0x64/0xe8 [ 280.014920]<0>-0)[<ffffffc0006e0f2c>] hal\_dma\_dump\_reg+0xf0/0xa0c [ 280.014932]<0>-0)[<ffffffc0006d934c>] btif\_bbs\_write.constprop.24+0x2b0/0x2dc [ 280.014943]<0>-0)[<ffffffc0006db1f8>] btif\_dma\_rx\_data\_receiver+0x1c/0x4c [ 280.014950]<0>-0)[<ffffffc0006e0cf8>] hal\_rx\_dma\_irq\_handler+0x170/0x2b4 [ 280.014959]<0>-0)[<ffffffc0006d7c6c>] btif\_rx\_dma\_irq\_handler+0x64/0xec [ 280.014970]<0>-0)[<ffffffc00011ff08>] handle\_irq\_event\_percpu+0x98/0x3c8 [ 280.014978]<0>-0)[<ffffffc000120280>] handle\_irq\_event+0x48/0x78 [ 280.014987]<0>-0)[<ffffffc0001230c4>] handle\_fasteoi\_irq+0xb0/0x150 [ 280.014996]<0>-0)[<ffffffc00011f784>] generic\_handle\_irq+0x30/0x4c [ 280.015005]<0>-0)[<ffffffc0000848e4>] handle\_IRQ+0x94/0x1cc [ 280.015012]<0>-0)[<ffffffc000081608>] gic\_handle\_irq+0x3c/0x80 |

**代码位置：**

kernel-3.18/mm/vmalloc.c

static struct vm\_struct \*\_\_get\_vm\_area\_node(unsigned long size,  
 unsigned long align, unsigned long flags, unsigned long start,  
 unsigned long end, int node, gfp\_t gfp\_mask, const void \*caller)  
{  
 struct vmap\_area \*va;  
 struct vm\_struct \*area;  
  
 BUG\_ON(in\_interrupt());  
 ......  
}

**问题解读：**

BUG\_ON放在这里意思是在**不允许在中断里调用\_\_get\_vm\_area\_node()函数**。

为什么呢？应该是设计时就不允许在中断里使用的，该函数使用了全局变量，用spin\_lock保护，这种锁无法保护到中断上下文。如果在中断里使用\_\_get\_vm\_area\_node()可能会引起一系列问题。

哪些常用函数会调用到\_\_get\_vm\_area\_node()呢？

* vmalloc()系列函数
* ioremap()系列函数，包括of\_iomap()函数

也即是说，这些函数禁止在中断里调用！

**问题解决：**

如果是在中断里调用ioremap()函数则可以考虑在初始化时就map好。比如上面的例子，就应该在初始化时就map好，在中断里直接用就行了。

如果是vmalloc()，可以考虑用kmalloc()替换。

20、BUG at dmp\_wd\_handler

**异常现场**：

在\_\_exp\_main.txt和SYS\_KERNEL\_LOG里看到如下log：

**死机位置在函数dpm\_wd\_handler()中，是这里的BUG()触发了Exception**

|  |
| --- |
| <4>[  341.525222] 2)**PC is at dpm\_wd\_handler**+0x2c/0x30  <7>[  328.440667] 2)[57] bus device\_suspend <7>[  328.440674] 2)dev->driver->name=mtk-msdc <3>[  328.440714] 2)msdc1 -> PM Suspend <4>[  328.440741] 2)msdc1 select card<0x00000000> <4>[  328.440776] 2)msdc LDO<2> power off <4>[  328.440796] 2)msdc LDO<3> power off <3>[  328.450820] 2)msdc1 -> set mclk to 0 <7>[  328.451843] 2)[58] bus device\_suspend <7>[  328.451850] 2)dev->driver->name=mtk-msdc <3>[  328.451863] 2)msdc0 -> PM Suspend <4>[  328.451880] 2)msdc0 select card<0x00000000> <4>[  328.451933] 2)msdc LDO<4> power off <7>[  328.451973] 2)[59] bus device\_suspend <7>[  328.451988] 2)[60] bus device\_suspend <7>[  328.452001] 2)[61] bus device\_suspend <7>[  328.452008] 2)dev->driver->name=akm8963  。。。。。。  <0>[  340.489412] 2)akm8963 2-000c: **\*\*\*\* DPM device timeout \*\*\*\*** <4>[  340.489421] 2)**Backtrace:**  。。。。。。 |

**代码位置：**

[kernel-3.10](http://10.16.20.15/l0mp8/xref/kernel-3.10/)**/**[drivers](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/)**/**[base](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/)**/**[power](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/)**/**[main.c](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c)

|  |
| --- |
| /\*\*  [411](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "411) \* dpm\_wd\_handler - Driver suspend / resume watchdog handler.  [412](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "412) \*  [413](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "413) \* Called when a driver has timed out suspending or resuming.  [414](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "414) \* There's not much we can do here to recover so BUG() out for  [415](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "415) \* a crash-dump  [416](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "416) \*/  [417](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "417)**static** **void** [**dpm\_wd\_handler**](http://10.16.20.15/l0mp8/s?refs=dpm_wd_handler&project=kernel-3.10)(**unsigned** **long** **[data](http://10.16.20.15/l0mp8/s?refs=data&project=kernel-3.10)**)  [418](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "418){  [419](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "419) **struct** [**dpm\_watchdog**](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c#dpm_watchdog) \***[wd](http://10.16.20.15/l0mp8/s?refs=wd&project=kernel-3.10)** = (**void** \*)[data](http://10.16.20.15/l0mp8/s?defs=data&project=kernel-3.10);  [420](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "420) **struct** [device](http://10.16.20.15/l0mp8/s?defs=device&project=kernel-3.10) \***[dev](http://10.16.20.15/l0mp8/s?refs=dev&project=kernel-3.10)** = [wd](http://10.16.20.15/l0mp8/s?defs=wd&project=kernel-3.10)->**[dev](http://10.16.20.15/l0mp8/s?refs=dev&project=kernel-3.10)**;  [421](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "421) **struct** [task\_struct](http://10.16.20.15/l0mp8/s?defs=task_struct&project=kernel-3.10) \***[tsk](http://10.16.20.15/l0mp8/s?refs=tsk&project=kernel-3.10)** = [wd](http://10.16.20.15/l0mp8/s?defs=wd&project=kernel-3.10)->**[tsk](http://10.16.20.15/l0mp8/s?refs=tsk&project=kernel-3.10)**;  [422](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "422)  [423](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "423) [dev\_emerg](http://10.16.20.15/l0mp8/s?defs=dev_emerg&project=kernel-3.10)([dev](http://10.16.20.15/l0mp8/s?defs=dev&project=kernel-3.10), "\*\*\*\* DPM device timeout \*\*\*\*\n");  [424](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "424) [show\_stack](http://10.16.20.15/l0mp8/s?defs=show_stack&project=kernel-3.10)([tsk](http://10.16.20.15/l0mp8/s?defs=tsk&project=kernel-3.10), [NULL](http://10.16.20.15/l0mp8/s?defs=NULL&project=kernel-3.10));  [425](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "425)  [426](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "426) [BUG](http://10.16.20.15/l0mp8/s?defs=BUG&project=kernel-3.10)();  [427](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "427)} |

**问题分析：**

系统在call每个Module驱动的suspend/resume callback function的时候，会设定一个timer 来监控回调函数的执行；如果回调函数长时间没有执行完毕，定时器函数会调用BUG()让系统挂掉，

所以在客制化回调函数时，要确保不能长时间阻塞，否则就会走到dpm\_wdt\_handler()函数，如上log，先打出"\*\*\*\* DPM device timeout \*\*\*\*"，接着会将相关驱动的**Backtrace**印出来

程序如何跑入dpm\_wdt\_handler() ？

在\_\_device\_suspend()函数中，可以看到在函数开始位置会调用dpm\_wd\_set(&wd,dev),此处会设置一个timer,在suspend函数结尾处，会调用dpm\_wd\_clear(&wd),用于清除timer ，在start timer 和clear timer 之间的code 是做device suspend 的动作，这个过程必须在一定时间内完成，否则就会导致timer到期，触发dpm\_wdt\_handler()

**问题解决：**

第一，根据上述log中Backtrace 找到是哪个device导致，如是自己添加的device驱动，特别是客制化的部分，需自己查看一下是否存在device suspend太久的情况。

第二，如这部分code没有做任何修改，请提交e-service并注明是哪部分驱动引起的问题，以便我们迅速处理问题。

21、BUG at dpm\_drv\_timeout

dpm\_drv\_timeout()和dpm\_wd\_handler()一样，请参考《BUG at dpm\_wd\_handler》章节

22、BUG at check\_bytes\_and\_report

**异常现场**：

当你在SYS\_KERNEL\_LOG里看到如下log，那么就属于BUG at check\_bytes\_and\_report一类问题了：

|  |
| --- |
| <6>[ 492.558572]-(0)[1163:system\_server]=============================================================================  <6>[ 492.558599]-(0)[1163:system\_server]**BUG** kmalloc-128 (Tainted: P W O ):**Poison overwritten** <6>[ 492.558621]-(0)[1163:system\_server]----------------------------------------------------------------------------- <6>[ 492.558621] <6>[ 492.558649]-(0)[1163:system\_server]**INFO: 0xffffffc0553a93e9-0xffffffc0553a93e9. First byte 0x69 instead of 0x6b** <6>[ 492.558685]-(0)[1163:system\_server]**INFO: Allocated in** alloc\_vmap\_area.isra.31+0x6c/0x394 age=525 cpu=0 pid=8631 <6>[ 492.558708]-(0)[1163:system\_server] alloc\_debug\_processing+0x184/0x194 <6>[ 492.558731]-(0)[1163:system\_server] \_\_slab\_alloc.isra.55.constprop.64+0x5ac/0x5e8 <6>[ 492.558751]-(0)[1163:system\_server] kmem\_cache\_alloc\_trace+0x11c/0x240 <6>[ 492.558772]-(0)[1163:system\_server] alloc\_vmap\_area.isra.31+0x68/0x394 <6>[ 492.558793]-(0)[1163:system\_server] \_\_get\_vm\_area\_node.isra.32+0x98/0x190 <6>[ 492.558814]-(0)[1163:system\_server] \_\_vmalloc\_node\_range+0x64/0x284 <6>[ 492.558833]-(0)[1163:system\_server] vmalloc+0x2c/0x38 <6>[ 492.558854]-(0)[1163:system\_server] write\_pmsg+0x58/0x148 <6>[ 492.558881]-(0)[1163:system\_server]**INFO: Freed in** rcu\_process\_callbacks+0x230/0x988 age=179 cpu=0 pid=3 <6>[ 492.558902]-(0)[1163:system\_server] free\_debug\_processing+0x1f4/0x328 <6>[ 492.558921]-(0)[1163:system\_server] \_\_slab\_free+0x29c/0x374 <6>[ 492.558939]-(0)[1163:system\_server] kfree+0x22c/0x270 <6>[ 492.558959]-(0)[1163:system\_server] rcu\_process\_callbacks+0x22c/0x988 <6>[ 492.558979]-(0)[1163:system\_server] \_\_do\_softirq+0xd8/0x36c <6>[ 492.558998]-(0)[1163:system\_server] run\_ksoftirqd+0x6c/0xec <6>[ 492.559020]-(0)[1163:system\_server] smpboot\_thread\_fn+0x1fc/0x2c8 <6>[ 492.559039]-(0)[1163:system\_server] kthread+0xd8/0xf0 <6>[ 492.559063]-(0)[1163:system\_server]**INFO: Slab** 0xffffffbe020a4cf8 objects=12 used=12 fp=0x (null) flags=0x0080 <6>[ 492.559085]-(0)[1163:system\_server]**INFO: Object** 0xffffffc0553a93c0 @offset=960 fp=0xffffffc0553a9640 <6>[ 492.559085] <6>[ 492.559116]-(0)[1163:system\_server]**Bytes** b4 ffffffc0553a93b0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZZZZZZZZZ <6>[ 492.559140]-(0)[1163:system\_server]**Object** ffffffc0553a93c0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk <6>[ 492.559165]-(0)[1163:system\_server]**Object** ffffffc0553a93d0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk <6>[ 492.559189]-(0)[1163:system\_server]**Object** ffffffc0553a93e0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 69 6b 6b 6b 6b 6b 6b kkkkkkkkkikkkkkk <6>[ 492.559213]-(0)[1163:system\_server]**Object** ffffffc0553a93f0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk <6>[ 492.559237]-(0)[1163:system\_server]**Object** ffffffc0553a9400: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk <6>[ 492.559261]-(0)[1163:system\_server]**Object** ffffffc0553a9410: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk <6>[ 492.559286]-(0)[1163:system\_server]**Object** ffffffc0553a9420: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk <6>[ 492.559310]-(0)[1163:system\_server]**Object** ffffffc0553a9430: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5 kkkkkkkkkkkkkkk. <6>[ 492.559333]-(0)[1163:system\_server]**Redzone** ffffffc0553a9440: bb bb bb bb bb bb bb bb ........ <6>[ 492.559358]-(0)[1163:system\_server]**Padding** ffffffc0553a94c0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZZZZZZZZZ <6>[ 492.559382]-(0)[1163:system\_server]**Padding** ffffffc0553a94d0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZZZZZZZZZ <6>[ 492.559407]-(0)[1163:system\_server]**Padding** ffffffc0553a94e0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZZZZZZZZZ <6>[ 492.559431]-(0)[1163:system\_server]**Padding** ffffffc0553a94f0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZZZZZZZZZ <6>[ 492.559457]-(0)[1163:system\_server]CPU: 0 PID: 1163 Comm: system\_server Tainted: P B W O 3.18.22+ #1 <6>[ 492.559474]-(0)[1163:system\_server]Hardware name: MT6755 (DT) <6>[ 492.559491]-(0)[1163:system\_server]Call trace: <2>[ 492.559514]-(0)[1163:system\_server][<ffffffc00008b970>] dump\_backtrace+0x0/0x15c <2>[ 492.559534]-(0)[1163:system\_server][<ffffffc00008badc>] show\_stack+0x10/0x1c <2>[ 492.559557]-(0)[1163:system\_server][<ffffffc000abb050>] dump\_stack+0x74/0xb8 <2>[ 492.559578]-(0)[1163:system\_server][<ffffffc0001c44fc>] print\_trailer+0x140/0x288 <2>[ 492.559600]-(0)[1163:system\_server][<ffffffc0001c471c>] check\_bytes\_and\_report+0xd8/0x10c <2>[ 492.559622]-(0)[1163:system\_server][<ffffffc0001c4968>] check\_object+0x1ac/0x220 <2>[ 492.559644]-(0)[1163:system\_server][<ffffffc0001c69a8>] alloc\_debug\_processing+0x110/0x194 <2>[ 492.559667]-(0)[1163:system\_server][<ffffffc0001c7330>] \_\_slab\_alloc.isra.55.constprop.64+0x5ac/0x5e8 <2>[ 492.559689]-(0)[1163:system\_server][<ffffffc0001c7958>] kmem\_cache\_alloc\_trace+0x11c/0x240 <2>[ 492.559711]-(0)[1163:system\_server][<ffffffc0000c1eac>] \_\_async\_schedule+0x34/0x1a8 <2>[ 492.559734]-(0)[1163:system\_server][<ffffffc0000c2048>] async\_schedule\_domain+0x8/0x14 <2>[ 492.559759]-(0)[1163:system\_server][<ffffffc0008da144>] dapm\_power\_widgets+0x618/0x8bc <2>[ 492.559783]-(0)[1163:system\_server][<ffffffc0008dc17c>] snd\_soc\_dapm\_stream\_event+0xbc/0x110 <2>[ 492.559805]-(0)[1163:system\_server][<ffffffc0008d04f8>] snd\_soc\_suspend+0x294/0x468 <2>[ 492.559829]-(0)[1163:system\_server][<ffffffc000393e18>] platform\_pm\_suspend+0x20/0x50 <2>[ 492.559852]-(0)[1163:system\_server][<ffffffc000399194>] dpm\_run\_callback+0x50/0x184 <2>[ 492.559873]-(0)[1163:system\_server][<ffffffc00039ae98>] \_\_device\_suspend+0x128/0x3cc <2>[ 492.559894]-(0)[1163:system\_server][<ffffffc00039b1e0>] dpm\_suspend+0xa4/0x358 <2>[ 492.559915]-(0)[1163:system\_server][<ffffffc00039b4fc>] dpm\_suspend\_start+0x68/0x78 <2>[ 492.559939]-(0)[1163:system\_server][<ffffffc0000fe8d8>] suspend\_devices\_and\_enter+0x98/0x2d0 <2>[ 492.559960]-(0)[1163:system\_server][<ffffffc0000fefe4>] pm\_suspend+0x3c4/0x5c8 <2>[ 492.559982]-(0)[1163:system\_server][<ffffffc0000fd5b0>] state\_store+0xb0/0xe0 <2>[ 492.560004]-(0)[1163:system\_server][<ffffffc0003270c4>] kobj\_attr\_store+0x10/0x24 <2>[ 492.560026]-(0)[1163:system\_server][<ffffffc00023996c>] sysfs\_kf\_write+0x3c/0x48 <2>[ 492.560049]-(0)[1163:system\_server][<ffffffc000238ef4>] kernfs\_fop\_write+0x10c/0x178 <2>[ 492.560070]-(0)[1163:system\_server][<ffffffc0001d0808>] vfs\_write+0x98/0x1b8 <2>[ 492.560091]-(0)[1163:system\_server][<ffffffc0001d0f40>] SyS\_write+0x40/0xa0 <2>[ 492.560124]-(0)[1163:system\_server]**BUG: failure at kernel-3.18/mm/slub.c:796/check\_bytes\_and\_report()!** <6>[ 492.561176]-(0)[1163:system\_server][klog]fault\_level=0x7(data abort(aarch64)), fault\_type=6, fault\_msg=level 2 translation fault <6>[ 492.561201]-(0)[1163:system\_server]Unable to handle kernel paging request at virtual address 0000dead <6>[ 492.561220]-(0)[1163:system\_server]pgd = ffffffc03c136000 <2>[ 492.561229]-(0)[1163:system\_server][0000dead] \*pgd=00000000802de003, \*pud=00000000802de003, \*pmd=0000000000000000 <6>[ 492.561283]-(0)[1163:system\_server][klog]fault already exists:0x7, ignore:0x2 <6>[ 492.561306]-(0)[1163:system\_server]Internal error: Oops: 96000046 [#1] PREEMPT SMP <2>[ 492.561316]-(0)[1163:system\_server]disable aee kernel api <6>[ 492.561350]-(0)[1163:system\_server]CPU: 0 PID: 1163 Comm: system\_server Tainted: P B W O 3.18.22+ #1 <6>[ 492.561369]-(0)[1163:system\_server]Hardware name: MT6755 (DT) <6>[ 492.561395]-(0)[1163:system\_server]task: ffffffc03b158000 ti: ffffffc02f018000 task.ti: ffffffc02f018000 <6>[ 492.561418]-(0)[1163:system\_server]**PC is at check\_bytes\_and\_report**+0x108/0x10c <6>[ 492.561441]-(0)[1163:system\_server]LR is at check\_bytes\_and\_report+0x100/0x10c |

**代码位置：**

kernel-3.18/mm/slub.c

static int check\_bytes\_and\_report(struct kmem\_cache \*s, struct page \*page, u8 \*object, char \*what, u8 \*start, unsigned int value, unsigned int bytes)  
{  
 u8 \*fault;  
 u8 \*end;  
  
 fault = memchr\_inv(start, value, bytes);  
 if (!fault)  
 return 1;  
  
 end = start + bytes;  
 while (end > fault && end[-1] == value)  
 end--;  
  
 slab\_bug(s, "%s overwritten", what);  
 pr\_err("INFO: 0x%p-0x%p. First byte 0x%x instead of 0x%x\n", fault, end - 1, fault[0], value);  
 print\_trailer(s, page, object);  
  
 BUG(); /\* 这里发生KE \*/  
 restore\_bytes(s, what, value, fault, end);  
 return 0;  
}

**问题解读：**

slub是kernel非常重要的内存分配器（详情请看：doc/Documentation/vm/slub.txt），是基于buddy system分配器之上再细化的。其接口就是kmalloc()/kfree()等一系列函数，在kernel里被广泛使用。

任何内存管理相关的问题，slub也会有。比如：

* use after free。通过kmalloc()申请内存，用完之后kfree()释放，但是后面又再使用了这块释放的内存。
* 内存踩坏。
  + 通过kmalloc()申请的内存，使用时超出了当时申请的尺寸，将不属于你的内存踩坏了。
  + 其他模块意外将slub空闲/使用中的内存踩坏。
* double free。对一块kmalloc()申请的内存连续调用2次kfree()函数。
* HW故障。比如DRAM/CPU不稳定，导致原有的内存发生跳变，bitflip。上面的例子就是发生了bitflip，6b跳变为69了。

如何检查slub里是否发生异常呢？这就需要额外的内存做守卫了，kernel已有这样的功能，只要打开CONFIG\_SLUB\_DEBUG，kfree()后的内存会被格式化成6b，还有red zone格式化成bb，padding为5a，还有每次申请和释放都会记录调用栈，记录谁申请/释放的。在每次的内存申请都会做相关检查，如果出现故障，就会抛出对应的log并主动调用BUG()

**问题解决：**

* use after free。需要通过slub记录的调用栈找凶手，检查代码逻辑才行。
* 内存踩坏。通过slub记录的调用栈或采用MMU保护，请参考：[FAQ14614]如何用MMU保护buddy system？
* double free。从slub记录的调用栈可以很明显看出。
* bitflip。直接要硬件交叉比对（CPU/DRAM），排查硬件故障，上面的log的例子直接HW交叉比对。

23、BUG at \_i2c\_deal\_result

**异常现场：**

在\_\_exp\_main.txt和SYS\_KERNEL\_LOG里看到如下log：

**死机位置在函数dpm\_wd\_handler()中，是这里的BUG()触发了Exception**

|  |
| --- |
| [ 21.758078]-(1)[155:bat\_thread\_kthr]**Kernel BUG** at c0660080 [verbose debug info unavailable] [ 21.758092]-(1)[155:bat\_thread\_kthr]Internal error: Oops - BUG: 0 [#1] PREEMPT SMP ARM [ 29.114643]-(1)[155:bat\_thread\_kthr]Modules linked in: [ 29.114666]-(1)[155:bat\_thread\_kthr]CPU: 1 PID: 155 Comm: bat\_thread\_kthr Tainted: G W 3.10.48+ #1 [ 29.114679]-(1)[155:bat\_thread\_kthr]task: df2ce000 ti: df810000 task.ti: df810000 [ 29.114692]-(1)[155:bat\_thread\_kthr]**PC is at \_i2c\_deal\_result**+0x314/0x358 [ 29.114705]-(1)[155:bat\_thread\_kthr]LR is at i2c\_release\_md32\_semaphore+0x40/0x9c |

**代码位置：**

[kernel-3.18/drivers/misc/mediatek/i2c/$platform/i2c.c](http://10.16.20.15/l0mp8/xref/kernel-3.10/)

static s32 \_i2c\_deal\_result(struct mt\_i2c\_t \*i2c)  
{  
 ......  
 if (!(tmo == 0 || atomic\_read(&i2c->trans\_err))) {  
 /\*Transfer success ,we need to get data from fifo \*/  
 if ((!i2c->dma\_en) && (i2c->op == I2C\_MASTER\_RD || i2c->op == I2C\_MASTER\_WRRD)) {  
 data\_size = (i2c\_readl(i2c, OFFSET\_FIFO\_STAT) >> 4) & 0x000F;  
 BUG\_ON(data\_size > i2c->msg\_len);  
 /\* I2CLOG("data\_size=%d\n",data\_size); \*/  
 while (data\_size--) {  
 ......  
 }  
 ......  
 }  
 ......  
}

**问题分析：**

出现这种问题的原因是I2C模块接收到的数据超出驱动的预期（多接收）。

原因基本上是硬件故障，比如受到干扰，共用I2C引起的冲突等等。

**问题解决：**

检查驱动对应的I2C是否挂载了其他设备，如果可以先拔除其他共用的I2C设备理清问题，然后再从驱动配置和硬件电路入手排查。

24、BUG at mt\_pmic\_wrap\_irq

**异常现场：**

当你在SYS\_KERNEL\_LOG里看到如下log，那么就属于BUG at mt\_pmic\_wrap\_irq一类问题了：

|  |
| --- |
| [ 46.553472]<0>-(0)[0:swapper/0]**[PWRAP] ERROR**,line=305 @@@@Timeout: elapse time10000307, start46543462771 , current46553463078, setting timer10000000 [ 46.553485]<0>-(0)[0:swapper/0]**[PWRAP] ERROR**,line=387 wait\_for\_state\_ready\_init timeout when waiting for idle [ 46.553496]<0>-(0)[0:swapper/0]**[PWRAP] ERROR**,line=673 \_pwrap\_wacs2\_nochk read fail,return\_value=9 [ 46.553506]<0>-(0)[0:swapper/0][PWRAP] PMIC HW CID = 0x0 [ 46.553516]<0>-(0)[0:swapper/0][PWRAP] INT flag 0x7ffffff9 [ 46.553532]<0>-(0)[0:swapper/0]Unable to handle kernel paging request at virtual address 0000dead ...... [ 46.554444]<0>-(0)[0:swapper/0]**Internal error: Oops:** 96000046 [#1] PREEMPT SMP ...... [ 53.502536]<0>-(0)[0:swapper/0]CPU: 0 PID: 0 Comm: swapper/0 Tainted: G W 3.10.72+ #1 [ 53.502549]<0>-(0)[0:swapper/0]task: ffffffc000e142f0 ti: ffffffc000e00000 task.ti: ffffffc000e00000 [ 53.502565]<0>-(0)[0:swapper/0]**PC is at mt\_pmic\_wrap\_irq**+0x13c/0x15c [ 53.502577]<0>-(0)[0:swapper/0]LR is at mt\_pmic\_wrap\_irq+0x12c/0x15c |

**代码位置：**

kernel-3.18/drivers/misc/mediatek/pmic\_wrap/$platform/pwrap\_hal.c

static irqreturn\_t mt\_pmic\_wrap\_irq(int irqno, void \*dev\_id)  
{  
 unsigned long flags = 0;  
  
 ......  
 pwrap\_dump\_all\_register();  
  
 /\* clear interrupt flag \*/  
 WRAP\_WR32(PMIC\_WRAP\_INT0\_CLR, 0xffffffff);  
 PWRAPREG("INT0 flag 0x%x\n", WRAP\_RD32(PMIC\_WRAP\_INT0\_EN));  
 if (10 == g\_wrap\_wdt\_irq\_count || 1 == g\_case\_flag)  
 BUG\_ON(1); /\* 这里发生KE \*/  
 ......  
}

**问题解读：**

对PMIC数据通信有硬件超时保护，如果超时表示已出问题，需要检查硬件。

**问题解决：**

检查PMIC硬件

25、调用kmalloc前关闭抢断引起KE

**问题背景：**

手机后台收集的异常db，其中有一类问题疑似软件问题。

**分析过程：**

用GAT解开db，并结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），利用工具分析（也可以参考FAQ13941），解析出来的调用栈如下：

|  |
| --- |
| 详细描述: 进程Binder\_9在原子上下文(关闭抢断)发起调度, 请从调用栈查找关抢断代码并修复  参考信息: MediaTek On-Line> Quick Start> 深入分析Linux kernel exception框架> 实例篇: 案例分析> BUG at \_\_schedule\_bug 异常时间: 68526.582303秒  == CPU信息 == 崩溃CPU信息: CPU4: 进程名: Binder\_9, 进程标识符(pid): 1523 本地调用栈: vmlinux \_\_schedule\_bug(prev=0xFFFFFFC00E049000) + 104    <kernel/sched/core.c:3074> vmlinux schedule\_debug() + 1472                                              <kernel/sched/core.c:3088> vmlinux \_\_schedule() + 1548                                                      <kernel/sched/core.c:3182> vmlinux schedule() + 36                                                              <kernel/sched/core.c:3279> vmlinux \_\_down\_write\_nested(subclass=0) + 112                    <lib/rwsem-spinlock.c:213> vmlinux \_\_down\_write() + 12                                                     <lib/rwsem-spinlock.c:225> vmlinux down\_write() + 8                                                           <kernel/rwsem.c:50> vmlinux zram\_bvec\_rw(zram=0xFFFFFFC0460CF000, bvec=0xFFFFFFC037C75C68, index=27300, offset=0, rw=1) + 56 <drivers/staging/zram/zram\_drv.c:900> vmlinux \_\_zram\_make\_request() + 268                                     <drivers/staging/zram/zram\_drv.c:955> vmlinux zram\_make\_request(bio=0xFFFFFFC037C75C00) + 400 <drivers/staging/zram/zram\_drv.c:1010> vmlinux generic\_make\_request(bio=0xFFFFFFC037C75C00) + 132 <block/blk-core.c:1844> vmlinux submit\_bio(rw=1, bio=0xFFFFFFC037C75C00) + 172 <block/blk-core.c:1936> vmlinux \_\_swap\_writepage(page=0xFFFFFFBC025F6290) + 552 <mm/page\_io.c:321> vmlinux swap\_writepage(page=0xFFFFFFBC025F6290, wbc=0xFFFFFFC06712B4A8) + 44 <mm/page\_io.c:250> vmlinux pageout(page=0xFFFFFFBC025F6290, mapping=0xFFFFFFC000E84688) + 368 <mm/vmscan.c:492> vmlinux shrink\_page\_list(page\_list=0xFFFFFFC06712B690, zone=0xFFFFFFC000F36640, sc=0xFFFFFFC06712B8E8, ttu\_flags=TTU\_UNMAP) + 800 <mm/vmscan.c:993> vmlinux shrink\_inactive\_list(lruvec=0xFFFFFFC000F36A88, sc=0xFFFFFFC06712B8E8, lru=LRU\_INACTIVE\_ANON) + 608 <mm/vmscan.c:1476> vmlinux shrink\_list() + 428 <mm/vmscan.c:1807> vmlinux shrink\_lruvec(lruvec=0xFFFFFFC000F36A88, sc=0xFFFFFFC06712B8E8) + 1100 <mm/vmscan.c:2154> vmlinux shrink\_zone(sc=0xFFFFFFC06712B8E8) + 60 <mm/vmscan.c:2323> vmlinux shrink\_zones() + 128 <mm/vmscan.c:2468> vmlinux do\_try\_to\_free\_pages(zonelist=0xFFFFFFC000F37900, sc=0xFFFFFFC06712B8E8, shrink=0xFFFFFFC06712B8D8) + 240 <mm/vmscan.c:2535> vmlinux try\_to\_free\_pages(zonelist=0xFFFFFFC000F37900, order=1, gfp\_mask=2118352, nodemask=0) + 236 <mm/vmscan.c:2766> vmlinux \_\_perform\_reclaim() + 52 <mm/page\_alloc.c:2332> vmlinux \_\_alloc\_pages\_direct\_reclaim() + 52 <mm/page\_alloc.c:2353> vmlinux \_\_alloc\_pages\_slowpath() + 700 <mm/page\_alloc.c:2626> vmlinux \_\_alloc\_pages\_nodemask() + 1144 <mm/page\_alloc.c:2845> vmlinux \_\_alloc\_pages() <include/linux/gfp.h:311> vmlinux alloc\_pages\_exact\_node() <include/linux/gfp.h:329> vmlinux alloc\_slab\_page() + 48 <mm/slub.c:1314> vmlinux allocate\_slab() + 76 <mm/slub.c:1336> vmlinux new\_slab(s=0xFFFFFFC047001840, flags=32976, node=-1) + 132 <mm/slub.c:1397> vmlinux new\_slab\_objects() + 104 <mm/slub.c:2171> vmlinux \_\_slab\_alloc(s=0xFFFFFFC047001840, gfpflags=32976) + 640 <mm/slub.c:2332> vmlinux slab\_alloc\_node() + 296 <mm/slub.c:2406> vmlinux slab\_alloc() + 296 <mm/slub.c:2446> vmlinux \_\_kmalloc(size=8192, flags=32976) + 372 <mm/slub.c:3278> vmlinux kmalloc() + 20 <include/linux/slub\_def.h:174> vmlinux kzalloc() + 20 <include/linux/slab.h:520> vmlinux cmdq\_rec\_realloc\_cmd\_buffer(handle=0xFFFFFFC0B1A5E800) + 36 <drivers/misc/mediatek/cmdq/cmdq\_record.c:51> vmlinux cmdq\_rec\_realloc\_cmd\_buffer() + 12 <drivers/misc/mediatek/cmdq/cmdq\_record.c:47> vmlinux cmdq\_append\_command(handle=0xFFFFFFC0B1A5E800, code=CMDQ\_CODE\_WRITE, argA=335634148, argB=1043319794) + 172 <drivers/misc/mediatek/cmdq/cmdq\_record.c:197> vmlinux cmdqRecWrite(handle=0xFFFFFFC0B1A5E800, value=1043319794, mask=4294967295) + 76 <drivers/misc/mediatek/cmdq/cmdq\_record.c:435> vmlinux disp\_gamma\_write\_lut\_reg(cmdq=0xFFFFFFC0B1A5E800, id=DISP\_GAMMA0, lock=0) + 452 <drivers/misc/mediatek/dispsys/mt6795/ddp\_gamma.c:110> vmlinux disp\_gamma\_set\_lut() + 136 <drivers/misc/mediatek/dispsys/mt6795/ddp\_gamma.c:153> vmlinux disp\_gamma\_io(module=DISP\_MODULE\_GAMMA, msg=1208252439) + 196 <drivers/misc/mediatek/dispsys/mt6795/ddp\_gamma.c:297> vmlinux dpmgr\_path\_user\_cmd(dp\_handle=0xFFFFFFC045E0F000, msg=1208252439, arg=547229475328, cmdqhandle=0xFFFFFFC0B1A5E800) + 380 <drivers/misc/mediatek/dispsys/mt6795/ddp\_manager.c:1124> vmlinux primary\_display\_user\_cmd() + 184 <drivers/misc/mediatek/videox/mt6795/primary\_display.c:4917> vmlinux mtk\_disp\_mgr\_ioctl() + 1008 <drivers/misc/mediatek/videox/mt6795/mtk\_disp\_mgr.c:2040> vmlinux disp\_unlocked\_ioctl() + 8 <drivers/misc/mediatek/videox/mt6795/mtk\_mira.c:21> vmlinux proc\_reg\_unlocked\_ioctl(file=0xFFFFFFC07209A100, arg=547229475328) + 76 <fs/proc/inode.c:252> vmlinux vfs\_ioctl() + 20 <fs/ioctl.c:43> vmlinux do\_vfs\_ioctl(filp=0xFFFFFFC07209A100, fd=310, cmd=1208252439, arg=547229475328) + 852 <fs/ioctl.c:598> vmlinux SYSC\_ioctl() + 112 <fs/ioctl.c:613> vmlinux SyS\_ioctl() + 128 <fs/ioctl.c:604> vmlinux cpu\_switch\_to() + 72 <arch/arm64/kernel/entry.S:673> == 栈结束 == |

这题是BUG at \_\_schedule\_bug，因此需要找出是谁关闭了抢断，需要从调用栈入手。

我们一一排查，不过前面一截都是kernel原生函数，不应该有问题，因此从cmdq\_rec\_realloc\_cmd\_buffer()函数开始排查，一一检查哪里关闭了抢断。

我们发现disp\_gamma\_set\_lut()函数有问题，代码如下：

static int disp\_gamma\_set\_lut(const DISP\_GAMMA\_LUT\_T \_\_user \*user\_gamma\_lut, void \*cmdq)  
{  
 ......  
 if (0 <= id && id < DISP\_GAMMA\_TOTAL) {  
 spin\_lock(&g\_gamma\_global\_lock);  
  
 old\_lut = g\_disp\_gamma\_lut[id];  
 g\_disp\_gamma\_lut[id] = gamma\_lut;  
  
 ret = disp\_gamma\_write\_lut\_reg(cmdq, id, 0);  
  
 spin\_unlock(&g\_gamma\_global\_lock);  
 ......  
}

可以看到在调用disp\_gamma\_write\_lut\_reg()函数前，调用了spin\_lock()，而spin\_lock()函数会关闭抢断的！！！

一般情况下spin lock包含的代码必须尽快执行完后释放掉，以免出现死锁等复杂问题。

而kmalloc()一般带有GFP\_KERNEL，可能会引起睡眠！这直接导致了这题的KE。

**根本原因：**

kmalloc(x, GFP\_KERNEL)会引起睡眠，不能在spin lock内使用！用其他可睡眠的锁或用GFP\_ATOMIC类型分配内存。

kernel开发的工程师要特别注意这个问题。

**解决方法：**

将spin\_lock换成mutex\_lock。

26、timer重复初始化引起KE

**问题背景：**

相机预览时黑屏死机重启。

**分析过程：**

用GAT解开db，并结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），利用工具分析（也可以参考FAQ13941），解析出来的调用栈如下：

|  |
| --- |
| 详细描述: Kernel panic{主动调用panic()}，请结合崩溃进程调用栈检查相关代码  异常时间: 4665.105173秒    == CPU信息 == 崩溃CPU信息: CPU2: 进程名: swapper/2, 进程标识符(pid): 0, 中断: 关 本地调用栈: vmlinux \_\_list\_add() + 120                                                   <lib/list\_debug.c:26> vmlinux list\_add\_tail() + 8                                                    <include/linux/list.h:76> vmlinux \_\_internal\_add\_timer(base=0xDF92C000, timer=0xC28E4470) + 124 <kernel/timer.c:381> vmlinux internal\_add\_timer(base=0xDF92C000, timer=0xC28E4470) + 28 <kernel/timer.c:386> vmlinux \_\_mod\_timer() + 184 <kernel/timer.c:769> vmlinux mod\_timer\_pinned(timer=0xC28E4470, expires=435638) + 228 <kernel/timer.c:892> vmlinux cpufreq\_interactive\_timer\_resched(pcpu=0xC28E4470) + 120 <drivers/cpufreq/cpufreq\_interactive.c:139> vmlinux cpufreq\_interactive\_idle\_end() + 236                              <drivers/cpufreq/cpufreq\_interactive.c:516> vmlinux cpufreq\_interactive\_idle\_notifier(nb=0xC0EAF98C, val=1, data=0) + 272 <drivers/cpufreq/cpufreq\_interactive.c:1114> vmlinux notifier\_call\_chain(nl=0xC0E863C0, val=1, v=0, nr\_to\_call=-1, nr\_calls=0xC0E88540) + 112 <kernel/notifier.c:128> vmlinux \_\_atomic\_notifier\_call\_chain(nr\_to\_call=-1, nr\_calls=0xDF8EE000) + 120 <kernel/notifier.c:234> vmlinux atomic\_notifier\_call\_chain(nh=0xC0E8639C, v=0) + 36 <kernel/notifier.c:243> vmlinux idle\_notifier\_call\_chain(val=1) + 32 <kernel/cpu.c:817> vmlinux arch\_cpu\_idle\_enter() + 24 <arch/arm/kernel/process.c:239> vmlinux cpu\_idle\_loop() + 100 <kernel/cpu/idle.c:92> vmlinux cpu\_startup\_entry(state=CPUHP\_ONLINE) + 164 <kernel/cpu/idle.c:147> vmlinux secondary\_start\_kernel() + 572 <arch/arm/kernel/smp.c:398> == 栈结束 == |

根据KE所在位置lib/list\_debug.c:26查看对应的代码：

void \_\_list\_add(struct list\_head \*new, struct list\_head \*prev, struct list\_head \*next)  
{  
 if (WARN(next->prev != prev, "list\_add corruption. next->prev should be prev (%p), but was %p. (next=%p).\n", prev, next->prev, next) ||  
 WARN(prev->next != next, "list\_add corruption. prev->next should be next (%p), but was %p. (prev=%p).\n", next, prev->next, prev) ||  
 WARN(new == prev || new == next, "list\_add double add: new=%p, prev=%p, next=%p.\n", new, prev, next))  
 BUG();  
 next->prev = new;  
 new->next = next;  
 new->prev = prev;  
 prev->next = new;  
}

查看log也印出了：

|  |
| --- |
| [ 4656.457111]-(2)[0:swapper/2]list\_add corruption. prev->next should be next (df92c5e4), but was df9685ec. (prev=de0b5180). |

初步判断是链表被踩坏，通过log无法进一步分析，需要借助trace32或gdb了，这里使用了trace32，通过脚本启动trace。发生异常的base的链表地址从log看到是0xDF92C5E4，发现：

|  |
| --- |
| 当时的base： base = 0xDF92C000 -> ( lock = (rlock = (raw\_lock = (slock = 4105303216, ti running\_timer = 0x0, timer\_jiffies = 435637, next\_timer = 435604, active\_timers = 6, tv1 = ( vec = ( ...... ( next = 0xDE0B5180 -> ( /\* 这个vec地址是：0xDF92C5E4 \*/ next = 0xDF9685EC, prev = 0xDF9685EC**/\* 应该要指向0xDF92C5E4，但却指向了0xDF9685EC \*/),** prev = 0xDE0B5180 -> ( next = 0xDF9685EC, prev = 0xDF9685EC)), (next = 0xDF92C5EC, prev = 0xDF92C5EC), ...... |

看到0xDF92C5E4的双向链表异常，next->prev != next了，怎么回事？我们再查看下0xDF9685EC是什么timer：

|  |
| --- |
| 查看0xDE0B5180的timer：  (struct timer\_list \*)0xDE0B5180 = 0xDE0B5180 -> ( entry = ( next = 0xDF9685EC, **/\* 另外1个timer\_base的链表!!!(通过trace32可以大致看出来) \*/** prev = 0xDF9685EC), expires = 435639, base = 0xDF968000, function = tmg399x\_rgbc\_poll\_handle, data = 0xDE0B5000, slack = -1, lockdep\_map = (key = 0xC16F64D4, class\_cache = (0x0, 0x0), name = 0xC0CC59C0 |

看到是tmg399x\_rgbc\_poll\_handle对应的timer，应该是被添加到了另1个timer base里的链表了，也就是这个timer有2个timer base指向它。

拿掉对应的源代码tmg399x.c，发现是chip->rgbc\_timer，检查其所有使用到该timer的代码：

static int tmg399x\_als\_enable(struct tmg399x\_chip \*chip, int on)  
{  
    ......  
    if (on) {  
        ......  
        setup\_timer(&chip->rgbc\_timer,   tmg399x\_rgbc\_poll\_handle, (unsigned long)chip);  
        chip->rgbc\_timer.expires = jiffies + HZ/10;  
        add\_timer(&chip->rgbc\_timer);  
        ......  
    } else {  
        del\_timer(&chip->rgbc\_timer);  
        ......  
    }  
    ......  
    return ret;  
}  
   
void tmg399x\_report\_als(struct tmg399x\_chip \*chip)  
{  
    ......  
    mod\_timer(&chip->rgbc\_timer, jiffies + HZ/10);  
    ......  
}  
   
void tmg399x\_get\_als(struct tmg399x\_chip \*chip)  
{  
    ......  
    mod\_timer(&chip->rgbc\_timer, jiffies + HZ/10);  
}  
   
static int tmg399x\_probe(struct i2c\_client \*client, const struct i2c\_device\_id \*idp)  
{  
    ......  
    init\_timer(&chip->rgbc\_timer);  
    ......  
    return ret;  
}

其中tmg399x\_als\_enable里的setup\_timer使用不当，这个只能在timer初始化时调用，而按源文件这样写，可能导致chip->rgbc\_timer还在其中1个timer base时，就重新对timer初始化，并且通过add\_timer添加到另外1个timer base里，导致原先的timer base超时时操作链表异常引起KE。

**根本原因：**

timer使用不当。

kernel开发的工程师要特别注意这个问题，timer初始化一次就够了。

**解决方法：**

将setup\_timer/add\_timer移动到tmg399x\_probe里，重新设计tmg399x\_als\_enable函数。

27、factory mode 关机 KE

**问题背景：**

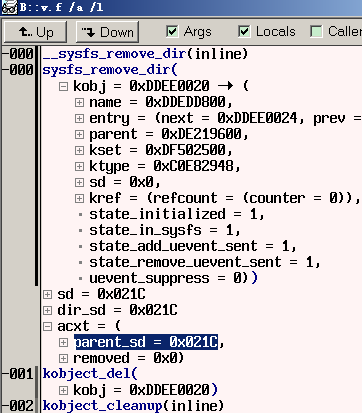
工厂模式测试完成后关机发生了KE。

**分析过程：**

用GAT解开db，并结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），利用工具分析（也可以参考FAQ13941），解析出来的调用栈如下：

|  |
| --- |
| 详细描述: 从错误的地址(0x00000260, 不在内核空间内)读数据，请结合崩溃进程调用栈检查相关代码  版本 : Linux version 3.10.48+ (robertyu@robertyu-server) (gcc version 4.8 (GCC) ) #1 SMP PREEMPT Mon Jan 19 15:48:30 CST 2015 异常时间: 111.555225秒  == CPU信息 == 崩溃CPU信息: CPU0: 进程名: factory, 进程标识符(pid): 205 内核调用栈: vmlinux rb\_first() + 12                                                     <lib/rbtree.c:422> vmlinux sysfs\_remove\_dir() + 80                                    <fs/sysfs/dir.c:839> vmlinux kobject\_del(参数1:0xDDEE0020, 参数4:15) + 20 <lib/kobject.c:514> vmlinux kobject\_release() + 104                                     <lib/kobject.c:567> vmlinux kobject\_put() + 76                                             <include/linux/kref.h:74> vmlinux device\_shutdown() + 112                                  <drivers/base/core.c:1192> vmlinux kernel\_power\_off() + 64                                    <kernel/sys.c:433> vmlinux sys\_reboot() + 380                                            <kernel/sys.c:540> == 栈结束 == 对应汇编指令: 行号 地址 指令 提示 lib/rbtree.c 419 : C0313FB4: MOV IP, SP C0313FB8: PUSH {FP, IP, LR, PC} C0313FBC: SUB FP, IP, #4 422 : C0313FC0: LDR R0, [R0] ; 进程停止在这里 423 : C0313FC4: CMP R0, #0 C0313FC8: LDMEQ SP, {FP, SP, PC} 当时的寄存器值: R0: 00000260, R1: 22222222, R2: 017E4000, R3: 00000000, R4: DDEE0020, R5: C0E82948, R6: 0000021C, R7: DDEE0020 R8: DDEE004C, R9: C0F12E1C, SL: 00000000, FP: DAED3DC4, IP: DAED3DC8, SP: DAED3DB8, LR: C0211260, PC: C0313FC0 |

这种需要借助工具才行，用trace32分析：



对应的代码如下：alps/kernel/fs/sysfs/dir.c

static void \_\_sysfs\_remove\_dir(struct sysfs\_dirent \*dir\_sd)  
{  
    struct sysfs\_addrm\_cxt acxt;  
    struct rb\_node \*pos;  
    if (!dir\_sd)  
        return;  
    pr\_debug("sysfs %s: removing dir\n", dir\_sd->s\_name);  
    sysfs\_addrm\_start(&acxt, dir\_sd);  
    pos = rb\_first(&dir\_sd->s\_dir.children);  
    ......  
}  
void sysfs\_remove\_dir(struct kobject \* kobj)  
{  
    struct sysfs\_dirent \*sd = kobj->sd;  
    spin\_lock(&sysfs\_assoc\_lock);  
    kobj->sd = NULL;  
    spin\_unlock(&sysfs\_assoc\_lock);  
    \_\_sysfs\_remove\_dir(sd);  
}

sd是错误的指针导致了KE，当时这个项目存在各种HW reboot/bitflip等问题，以为这个也是同样的例子，怀疑硬件故障。

但后来一直在回报同样的问题，看起来更像是SW引起的。

拿到故障手机和bin，当时是有复现出1次，后面添加MMU protect buddy system（FAQ14614），理论上野指针应该比较容易抓到才对。但还是复现了，并且是老地方！

于是重新检查代码：发现几次异常的位置的dev都属于cooling\_device25，cooling\_devicexx，应该和thermal相关。拿到thermal代码，检查是否存在改动，发现没有。

根据cooling\_device25，找到create的位置：alps/kernel/drivers/thermal/thermal\_sys.c

struct device {  
    struct device  \*parent;  
    struct device\_private \*p;  
    struct kobject kobj; /\* <== \*/  
    ......  
}  
   
struct thermal\_cooling\_device {  
    int id;  
    char type[THERMAL\_NAME\_LENGTH];  
    struct device device;  
    void \*devdata;  
    const struct thermal\_cooling\_device\_ops \*ops;  
    struct list\_head node;  
};  
   
struct thermal\_cooling\_device \*thermal\_cooling\_device\_register(char \*type, void \*devdata, const struct thermal\_cooling\_device\_ops \*ops)  
{  
    ......  
    cdev = kzalloc(sizeof(struct thermal\_cooling\_device), GFP\_KERNEL);  
    if (!cdev)  
        return ERR\_PTR(-ENOMEM);  
    ......  
}

看到是通过kzalloc申请的，落在objsize为1024的slub里，当时的想法是，既然老是踩坏这里，那就在这里加guard page！代码如下（必须搭配MMU protect buddy system）：alps/kernel/mm/slub.c

 tatic inline struct page \*alloc\_slab\_page(struct kmem\_cache \*s, gfp\_t flags, int node, struct kmem\_cache\_order\_objects oo)  
{  
    int order = oo\_order(oo);  
    /\* add this block \*/  
    struct page \*page;  
    int EnProt = 0;  
       
    if (s->objsize == 1024 && order == 0) {  
        order = 1;  
        EnProt = 1;  
    }  
    /\* end block \*/  
    flags |= \_\_GFP\_NOTRACK;  
    if (node == NUMA\_NO\_NODE)  
#ifndef CONFIG\_MTK\_PAGERECORDER  
        page = alloc\_pages(flags, order);  
#else  
        page = alloc\_pages\_nopagedebug(flags, order);  
#endif  
    else  
        page = alloc\_pages\_exact\_node(node, flags, order);  
    /\* add this block \*/  
#ifdef CONFIG\_DEBUG\_RODATA  
    if (EnProt && page && !PageHighMem(page)) {  
        set\_memory\_invalid((unsigned long)lowmem\_page\_address(page), 1);  
        page += 1;  
    }  
#endif  
    return page;  
    /\* end block \*/  
}

重新复制问题，结果反馈没有在关机时重启了，而在测试camera时崩溃了。马上分析log：

|  |
| --- |
| 详细描述: 写数据**0x21C**到没有写权限的地址(**0xDE24C038**)，请结合崩溃进程调用栈检查相关代码  异常时间: 60.486207秒  == CPU信息 == 崩溃CPU信息: CPU0: 进程名: factory, 进程标识符(pid): 222 内核调用栈: vmlinux primary\_display\_config\_input\_multiple(参数1:\_\_func\_\_.34346) + 536 <mediatek/platform/mt6752/kernel/drivers/videox/primary\_display.c:2617> vmlinux mtkfb\_ioctl(参数1:0xDE217000, 参数3:0xB710A9B8) + 368 <mediatek/platform/mt6752/kernel/drivers/videox/mtkfb.c:1509> vmlinux do\_fb\_ioctl() + 940 <drivers/video/fbmem.c:1238> vmlinux fb\_ioctl(参数1:0xDB67A000, 参数3:0xB710A9B8) + 72 <drivers/video/fbmem.c:1252> vmlinux do\_vfs\_ioctl(参数1:0xDB67A000, 参数2:24, 参数3:0x40744F02, 参数4:0xB710A9B8) + 1028 <fs/ioctl.c:43> vmlinux sys\_ioctl() + 124 <fs/ioctl.c:613> == 栈结束 == 对应汇编指令: 行号 地址 指令 提示 mediatek/platform/mt6752/kernel/drivers/videox/primary\_display.c 5451: C0751218: LDR R3, [R2, #4] C075121C: ADD R1, R1, #1 C0751220: STR R1, [FP, #-56] C0751224: CMP R1, R3 C0751228: BCS C0751414 5456: C075122C: LDR R7, [R4, #8] 2608: C0751230: MOV R0, #0 5457: C0751234: LDR R1, [FP, #-60] 2608: C0751238: STR R0, [FP, #-52] 5457: C075123C: RSB R3, R7, R7, LSL #4 2609: C0751240: STR R0, [FP, #-48] 5457: C0751244: ADD R3, R1, R3, LSL #3 2610: C0751248: CMN R3, #4 C075124C: CMPNE R8, R0 C0751250: ADD SL, R3, #4 C0751254: BEQ C075197C 2617: C0751258: STR R7, [SL, #4] ; 进程停止在这里 2618: C075125C: MOV R2, #1 C0751260: STR R2, [SL, #96] 当时的寄存器值: R0: 00000000, R1: DE23C310, R2: C1850508, R3: DE24C030, R4: C185079C, R5: 00000000, R6: 00000001, R7: 0000021C R8: C18507A4, R9: C1855DD0, SL: DE24C034, FP: DB75DB3C, IP: DB75DAE0, SP: DB75DAE0, LR: C07511CC, PC: C0751258 |

根据代码推导，结果如下：原因是mtkfb\_ioctl()函数的MTKFB\_SET\_VIDEO\_LAYERS里从native传下来的layer有问题，导致session\_input->config[5]赋值错误！！！！！

具体在alps/mediatek/platform/mt6752/kernel/drivers/videox/primary\_display.c

 static int \_convert\_disp\_input\_to\_ovl(OVL\_CONFIG\_STRUCT \*dst, disp\_input\_config \*src)  
{  
    int ret;  
    unsigned int Bpp = 0;  
    unsigned int bpp = 0;  
       
    if(!src || !dst)  
        return -1;  
    dst->layer = src->layer\_id; /\* 这里发生异常，dst指向错误的地址！ \*/  
    ......  
}  
   
static int \_config\_ovl\_input(disp\_session\_input\_config \*session\_input,  
                    disp\_path\_handle disp\_handle,  
                    cmdqRecHandle cmdq\_handle )  
{  
    ......  
    layer = input\_cfg->layer\_id; /\* layer = 540，超出8个layer限制！ \*/  
    ovl\_cfg = &(data\_config->ovl\_config[layer]);  
    \_convert\_disp\_input\_to\_ovl(ovl\_cfg, input\_cfg);  
    ......  
}

用trace32查看结构体：

|  |
| --- |
| session\_input = 0xC1850508 -> ( session\_id = 0, config\_layer\_num = 6, config = ( (layer\_id = 1, layer\_enable = 0, buffer\_source = DISP\_BUFFER\_ION, src\_base\_addr = 0x0, src\_phy\_addr = 0x0, src\_direct\_link = 0, src\_fmt = DISP\_FORMAT\_UNKNOWN, src\_ (layer\_id = 0, layer\_enable = 0, buffer\_source = DISP\_BUFFER\_ION, src\_base\_addr = 0x0, src\_phy\_addr = 0x0, src\_direct\_link = 0, src\_fmt = DISP\_FORMAT\_UNKNOWN, src\_ (layer\_id = 0, layer\_enable = 0, buffer\_source = DISP\_BUFFER\_ION, src\_base\_addr = 0x0, src\_phy\_addr = 0x0, src\_direct\_link = 0, src\_fmt = DISP\_FORMAT\_UNKNOWN, src\_ (layer\_id = 0, layer\_enable = 0, buffer\_source = DISP\_BUFFER\_ION, src\_base\_addr = 0x0, src\_phy\_addr = 0x0, src\_direct\_link = 0, src\_fmt = DISP\_FORMAT\_UNKNOWN, src\_ (layer\_id = 0, layer\_enable = 0, buffer\_source = DISP\_BUFFER\_ION, src\_base\_addr = 0x0, src\_phy\_addr = 0x0, src\_direct\_link = 0, src\_fmt = DISP\_FORMAT\_UNKNOWN, src\_ ( layer\_id = 540, **/\* 错误的layer\_id！！！！！感觉像是width或height \*/** layer\_enable = 0, buffer\_source = DISP\_BUFFER\_ION, src\_base\_addr = 0x0, src\_phy\_addr = 0x0, src\_direct\_link = 0, src\_fmt = DISP\_FORMAT\_UNKNOWN, **/\* 错误的fmt！！！！！ \*/** src\_use\_color\_key = 0, src\_color\_key = 0, src\_pitch = 0, src\_offset\_x = 0, src\_offset\_y = 0, src\_width = 0, src\_height = 0, tgt\_offset\_x = 0, tgt\_offset\_y = 0, tgt\_width = 0, tgt\_height = 0, layer\_rotation = DISP\_ORIENTATION\_0, layer\_type = DISP\_LAYER\_2D, video\_rotation = DISP\_ORIENTATION\_0, isTdshp = 0, next\_buff\_idx = 0, identity = 0, connected\_type = 0, security = DISP\_NORMAL\_BUFFER, alpha\_enable = 0, alpha = 0, sur\_aen = 0, src\_alpha = DISP\_ALPHA\_ONE, dst\_alpha = DISP\_ALPHA\_ONE, frm\_sequence = 0, yuv\_range = DISP\_YUV\_BT601\_FULL), (layer\_id = 0, layer\_enable = 0, buffer\_source = DISP\_BUFFER\_ION, src\_base\_addr = 0x0, src\_phy\_addr = 0x0, src\_direct\_link = 0, src\_fmt = DISP\_FORMAT\_UNKNOWN, src\_ (layer\_id = 0, layer\_enable = 0, buffer\_source = DISP\_BUFFER\_ION, src\_base\_addr = 0x0, src\_phy\_addr = 0x0, src\_direct\_link = 0, src\_fmt = DISP\_FORMAT\_UNKNOWN, src\_ |

static int \_convert\_fb\_layer\_to\_disp\_input(struct fb\_overlay\_layer\* src, disp\_input\_config \*dst)而session\_input内容从哪里来的？看代码alps/mediatek/platform/mt6752/kernel/drivers/videox/mtkfb.c：

{  
    dst->layer\_id = src->layer\_id;  
    ......  
    switch (src->src\_fmt) {  
        ......  
        default:  
            DISPERR("Invalid color format: 0x%x\n", src->src\_fmt);  
            return -1;  
    }  
    ......  
}  
   
static int mtkfb\_ioctl(struct fb\_info \*info, unsigned int cmd, unsigned long arg)  
{  
    ......  
    switch (cmd)  
    {        
    case MTKFB\_SET\_OVERLAY\_LAYER:  
    {  
        struct fb\_overlay\_layer layerInfo;  
        disp\_input\_config \*input;  
           
        copy\_from\_user(&layerInfo, (void \_\_user \*)arg, sizeof(layerInfo));  
        ......  
        memset((void\*)&session\_input, 0, sizeof(session\_input));  
        input = &session\_input.config[session\_input.config\_layer\_num++];  
        \_convert\_fb\_layer\_to\_disp\_input(&layerInfo, input);  
        primary\_display\_config\_input\_multiple(&session\_input);  
        primary\_display\_trigger(1, NULL, 0);  
        return (r);  
    }  
    ......  
    }  
}

这也结合了log为何有以下错误的msg印出：

|  |
| --- |
| [ 60.455067] (0)[222:factory][DISP][\_convert\_fb\_layer\_to\_disp\_input #475]ERROR:Invalid color format: 0x3c0 |

剩下就需要追查上层为何传错误的结构体信息下来了！

**根本原因：**

上层传错误的disp信息，导致数组越界溢出！

**解决方法：**

修复上层disp代码，同时驱动做边界检查！

28、变量没有锁保护

**问题背景：**

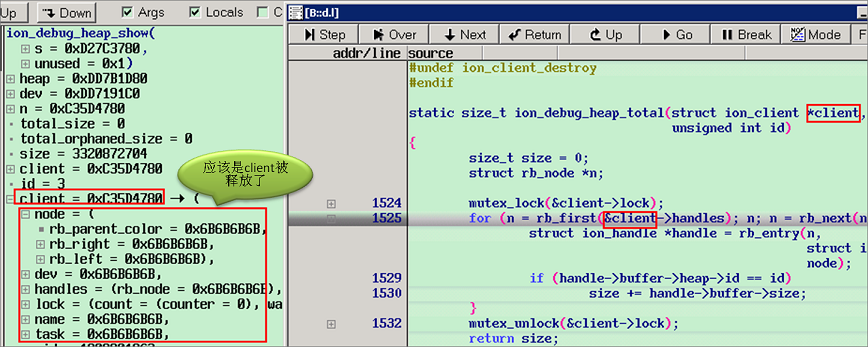
跑monkey跑出KE。

**分析过程：**

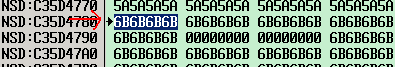
用GAT解开db，查看db.14.dbg/SYS\_KERNEL\_LOG：

|  |
| --- |
| <1>[58586.202903] (0)[10683:aee\_dumpstate]**Unable to handle kernel paging request at virtual address 6b6b6b73** ...... <4>[58586.204622] (1)[10683:aee\_dumpstate][<c028d508>] (rb\_first+0x0/0x30) from [<c02bf1f4>] (ion\_debug\_heap\_show+0x80/0x2a0) <4>[58586.204642] (1)[10683:aee\_dumpstate][<c02bf174>] (ion\_debug\_heap\_show+0x0/0x2a0) from [<c0172538>] (seq\_read+0x1c0/0x4a4) <4>[58586.204662] (1)[10683:aee\_dumpstate][<c0172378>] (seq\_read+0x0/0x4a4) from [<c0151e80>] (vfs\_read+0xac/0x13c) <4>[58586.204681] (1)[10683:aee\_dumpstate][<c0151dd4>] (vfs\_read+0x0/0x13c) from [<c0151f54>] (sys\_read+0x44/0x70) ...... <4>[58586.204833]-(1)[10683:aee\_dumpstate]PC is at rb\_first+0x20/0x30 <4>[58586.204844]-(1)[10683:aee\_dumpstate]LR is at ion\_debug\_heap\_show+0x80/0x2a0 <4>[58586.204856]-(1)[10683:aee\_dumpstate]pc : [<c028d528>] lr : [<c02bf1f4>] psr: 20000013 <4>[58586.204863]-(1)[10683:aee\_dumpstate]sp : c5f07e78 ip : c5f07e88 fp : c5f07e84 <4>[58586.204873]-(1)[10683:aee\_dumpstate]r10: 00000003 r9 : c35d4794 r8 : c5f07f00 <4>[58586.204884]-(1)[10683:aee\_dumpstate]r7 : dd7191c0 r6 : d27c3780 r5 : dd7b1d80 r4 : c35d4780 <4>[58586.204895]-(1)[10683:aee\_dumpstate]r3 : 00000000 r2 : 00d9b000 r1 : 00000000 r0 : 6b6b6b6b <4>[58586.204907]-(1)[10683:aee\_dumpstate]Flags: nzCv IRQs on FIQs on Mode SVC\_32 ISA ARM Segment user <4>[58586.204919]-(1)[10683:aee\_dumpstate]Control: 10c5387d Table: 9279c06a DAC: 00000015 |

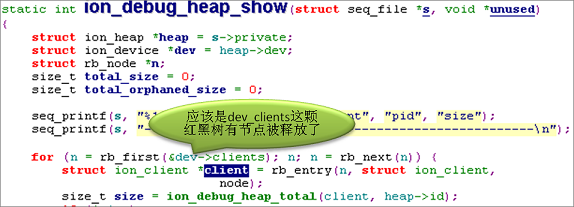
地址0x6b6b6b73，感觉像是slub free object填充字节，拿到vmlinux，建立trace32 debug环境：



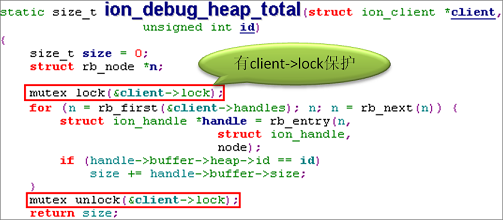
查看client所在地址：

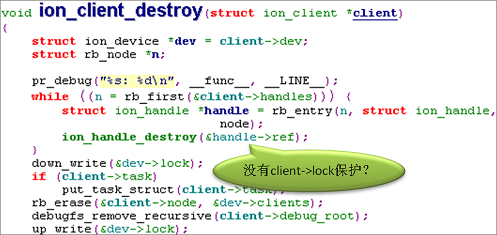


可以明显看到，刚好只向object头部，所以证实了client被释放的猜测。  
接着查看client从哪里来，看到代码：



所以是红黑树节点出现问题了，查看ion.c的逻辑比较简单，检查所有对这颗红黑树的操作（搜索dev->clients关键字），需要特别关注的是删除操作，结果发现：





所以很明显，在删除时没有锁保护导致变量被释放了。

**根本原因：**

变量没有锁保护导致的race condition。

**解决方法：**

请对应工程师修复。

PS：如果那块释放的object又被分配出去，那就非常麻烦了。

29、通话中频繁亮暗屏KE

**问题背景：**

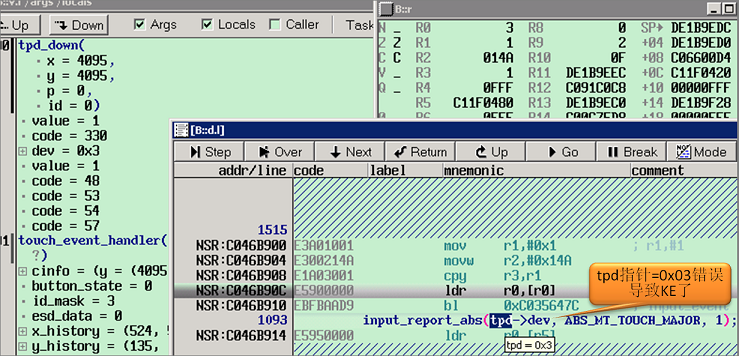
通话中频繁的通过P-senor亮暗屏幕，手机会重启。

**分析过程：**

用GAT解开db，并结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），查看log如下（SYS\_KERNEL\_LOG）：

|  |
| --- |
| <0>[  289.951182]-(1)[74:mtk-tpd]Internal error: Oops: 5 [#1] PREEMPT SMP ARM <4>[  289.955334]-(1)[74:mtk-tpd]CPU: 1    Not tainted  (3.4.5 #3) <4>[  289.956039]-(1)[74:mtk-tpd]PC is at tpd\_down+0x70/0x208 <4>[  289.956691]-(1)[74:mtk-tpd]LR is at trace\_hardirqs\_on\_caller+0x11c/0x224 <4>[  289.957526]-(1)[74:mtk-tpd]pc : [<c046b90c>]    lr : [<c00c75d8>]    psr: 60000013 <4>[  289.957533]-(1)[74:mtk-tpd]sp : de1b9ec0  ip : c091c0c8  fp : de1b9eec <4>[  289.959283]-(1)[74:mtk-tpd]r10: 0000000f  r9 : 00000002  r8 : 00000000 <4>[  289.960097]-(1)[74:mtk-tpd]r7 : 00000000  r6 : 00000fff  r5 : c11f0480  r4 : 00000fff <4>[  289.961074]-(1)[74:mtk-tpd]r3 : 00000001  r2 : 0000014a  r1 : 00000001  r0 : 00000003 |

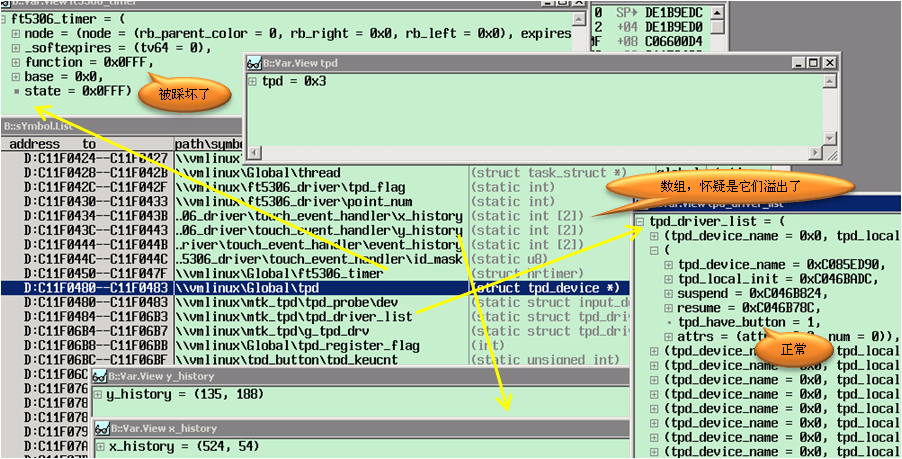
搭建trace32环境，查看对应的汇编如下：



对应的代码如下：ft5306\_driver.c

static  void tpd\_down(int x, int y, int p, int id) {  
    // input\_report\_abs(tpd->dev, ABS\_PRESSURE, p);  
//zxs 20130824  
TPD\_DMESG("TPD\_X=%d,TPD\_Y=%d\n",TPD\_X\_RES,TPD\_Y\_RES);  
TPD\_DMESG("x=%d,y=%d\n",x,y);  
if(0 == strncmp(MTK\_LCM\_PHYSICAL\_ROTATION, "180", 3))  
    {  
       if(y<=TPD\_Y\_RES)  
         {  
            x=TPD\_X\_RES-x;  
            y=TPD\_Y\_RES-y;  
         }   
   }  
TPD\_DMESG("rotation\_x=%d,rotation\_y=%d\n",x,y);  
//end    
     input\_report\_key(tpd->dev, BTN\_TOUCH, 1); /\* 这里崩溃的！！！ \*/  
     input\_report\_abs(tpd->dev, ABS\_MT\_TOUCH\_MAJOR, 1);  
     input\_report\_abs(tpd->dev, ABS\_MT\_POSITION\_X, x);  
     input\_report\_abs(tpd->dev, ABS\_MT\_POSITION\_Y, y);

查看到tpd是在mediatek/custom/common/kernel/touchpanel/src/mtk\_tpd.c里定义的指针，而这只文件基本没有修改过，应该是没有逻辑问题的。  
于是检查多份log，发现异常的位置都是固定的，并且踩坏的内容也是0x03。  
比较怀疑是自己修改导致，于是检查版本，看什么时候导致此问题的。  
这边同时检查tpd所在地址的上下变量查看是否正常，如下：



踩坏的范围较小，比较怀疑是数组溢出了，于是查看使用到这些数组的代码，发现（ft5306\_driver.c）：

static int tpd\_touchinfo(struct touch\_info \*cinfo, struct touch\_info \*pinfo)  
{  
......  
    i2c\_smbus\_read\_i2c\_block\_data(i2c\_client, 0x00, 8, &(data[0]));  
    i2c\_smbus\_read\_i2c\_block\_data(i2c\_client, 0x08, 8, &(data[8]));  
    i2c\_smbus\_read\_i2c\_block\_data(i2c\_client, 0x10, 8, &(data[16]));  
    i2c\_smbus\_read\_i2c\_block\_data(i2c\_client, 0x18, 8, &(data[24])); /\* i2c填充data数组 \*/  
......  
    for(i = 0; i <point\_num; i++)  
    {  
        cinfo->p[i] = (data[3+6\*i] >> 6); //event flag  
        cinfo->id[i] = (data[5+6\*i] >> 4); //id，data赋值给id数组，但是数据来源是i2c，不能保证小于2！！！  
......  
}  
   
static int touch\_event\_handler(void \*unused)  
{  
......  
    static int x\_history[CFG\_MAX\_TOUCH\_POINTS]; /\* #define CFG\_MAX\_TOUCH\_POINTS 2 \*/  
    static int y\_history[CFG\_MAX\_TOUCH\_POINTS];  
    static int event\_history[CFG\_MAX\_TOUCH\_POINTS];  
......  
    if (tpd\_touchinfo(&cinfo, &pinfo)) {  
    TPD\_DEBUG("point\_num = %d\n",point\_num);  
......  
        cur\_mask |= ( 1 << cinfo.id[idx] );  
        x\_history[cinfo.id[idx]]=cinfo.x[idx]; /\* 下标可能超出数组范围！！！ \*/  
        y\_history[cinfo.id[idx]]=cinfo.y[idx]; /\* 下标可能超出数组范围！！！ \*/  
        event\_history[cinfo.id[idx]] =cinfo.p[idx]; /\* 下标可能超出数组范围！！！ \*/  
......  
}

马上验证检查，最后发现就是这里的数组溢出导致。

**根本原因：**

x\_history、y\_history、event\_history数组溢出踩坏内存引起崩溃。

**解决方法：**

修正tp驱动。

30、work没有初始化就是用引起KE

**问题背景：**

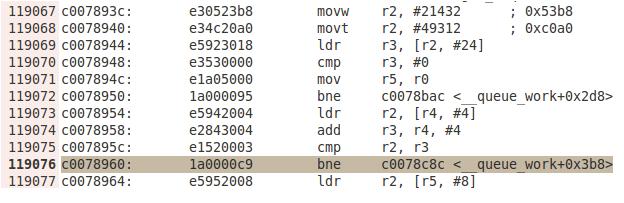
在弱光的环境下，开启闪光灯拍照，手机死机重启。

**分析过程：**

用GAT解开db，并结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），先查看db里的\_\_exp\_main.txt，发现是KE，这时再查看SYS\_KERNEL\_LOG, log如下（SYS\_KERNEL\_LOG）：

|  |
| --- |
| <2>[  506.632801]-(0)[0:swapper/0]Kernel BUG at c0078c8c [verbose debug info unavailable] <0>[  506.632835]-(0)[0:swapper/0]Internal error: Oops - BUG: 0 [#1] PREEMPT SMP ARM <4>[  506.632867]-(0)[0:swapper/0]Modules linked in: wlan\_mt6628 bf1ea000  ccci bf1bf000  ccci\_plat bf192000  mtk\_wmt\_wifi bf18e000  mtk\_fm\_drv bf161000  mtk\_stp\_bt bf153000  mtk\_stp\_gps bf146000  mtk\_stp\_uart bf13d000  mtk\_stp\_wmt bf0ba000  mtk\_hif\_sdio bf0a4000  devinfo bf0a0000  devapc bf099000  sec bf077000  vcodec\_kernel\_driver bf066000  mtklfb bf05b000  pvrsrvkm bf000000 <4>[  506.633140]-(0)[0:swapper/0]CPU: 0    Tainted: G        W     (3.4.5 #1) <4>[  506.633181]-(0)[0:swapper/0]PC is at \_\_queue\_work+0x3b8/0x4c8 <4>[  506.633213]-(0)[0:swapper/0]LR is at \_\_queue\_work+0x68/0x4c8 <4>[  506.633248]-(0)[0:swapper/0]pc : [<c0078c8c>]    lr : [<c007893c>]    psr: 00000193 <4>[  506.633265]-(0)[0:swapper/0]sp : c0981c38  ip : c0981c38  fp : c0981c6c <4>[  506.633300]-(0)[0:swapper/0]r10: 00000003  r9 : c22d0b80  r8 : c22d0940 <4>[  506.633332]-(0)[0:swapper/0]r7 : 00000000  r6 : 00000000  r5 : c22d6100  r4 : c11f9d80 <4>[  506.633366]-(0)[0:swapper/0]r3 : c11f9d84  r2 : 00000000  r1 : dd833000  r0 : c22d6100 <4>[  506.633402]-(0)[0:swapper/0]Flags: nzcv  IRQs off  FIQs on  Mode SVC\_32  ISA ARM  Segment kernel <4>[  506.633439]-(0)[0:swapper/0]Control: 10c5387d  Table: 98dfc06a  DAC: 00000015 |

BUG at c0078c8c :可以知道是直接调用BUG() trigger ke重启.  使用arm-linux-androideabi-objdump将vmlinux反编译出来:



搜索c0078c8c ,  是从0xc0078960跳过去的,  使用trace32加载vmlinux看到的结果:

|  |
| --- |
| \_NSD:C0078954|E5942004\_\_\_\_\_\_\_\_\_\_ldr\_\_\_\_r2,[r4,#0x4] NSD:C0078958|E2843004 add r3,r4,#0x4 ; r3,work,#4 NSD:C007895C|E1520003 cmp r2,r3 NSD:C0078960|1A0000C9 bne 0xC0078C8C 从这里可以知道R4 c11f9d80为work, 其成员的value为: \_\_address\_\_|\_\_\_\_\_\_0\_\_\_\_\_\_4\_\_\_\_\_\_8\_\_\_\_\_\_\_C\_0123456789ABCDEF NSD:C11F9D80|>00000001(data) 00000000(entry->next) 00000000(entry->prev) 00000000(func) ................ NSD:C11F9D90| 00000000 00000000 00000000 00000000 ................ NSD:C11F9DA0| 00000000 00000001 00000000 00000000 ................ NSD:C11F9DB0| 00000190 00000000 C11F9DB8 00000000 ................ |

对照C代码:  去判断work->entry是否为NULL, 即work->entry+0x04 是否等于work->entry->next,  条件不成立导致bug.

static void \_\_queue\_work(unsigned int cpu, struct workqueue\_struct \*wq, struct work\_struct \*work)  
{  
 ......  
  BUG\_ON(!list\_empty(&work->entry));  
 ......  
}

从work变量的value来看, 有2种可能性:

1. work被踩。
2. work没有初始化。

从反馈来看每次都必现, 比较怀疑是work没有初始化就去queue\_work。从backtrace知道是ledTimeOutCallback调用的queuework：

|  |
| --- |
| <4>[  506.647360]-(0)[0:swapper/0][<c00788d4>] (\_\_queue\_work+0x0/0x4c8) from [<c0078e10>] (queue\_work\_on+0x44/0x4c) <4>[  506.647406]-(0)[0:swapper/0][<c0078dcc>] (queue\_work\_on+0x0/0x4c) from [<c0078e68>] (queue\_work+0x2c/0x60) <4>[  506.647441]-(0)[0:swapper/0] r6:c22d0bd8 r5:dd833000 r4:c11f9d80 r3:00010003 <4>[  506.647527]-(0)[0:swapper/0][<c0078e3c>] (queue\_work+0x0/0x60) from [<c0078ebc>] (schedule\_work+0x20/0x24) <4>[  506.647562]-(0)[0:swapper/0] r5:00000000 r4:c11f9db8 <4>[  506.647625]-(0)[0:swapper/0][<c0078e9c>] (schedule\_work+0x0/0x24) from [<c044ba04>] (ledTimeOutCallback+0x18/0x20) |

拿到对应的source code：

enum hrtimer\_restart ledTimeOutCallback(struct hrtimer \*timer)  
{  
  PK\_DBG("ledTimeOut\_callback\n");  
 schedule\_work(&workTimeOut);  
    return HRTIMER\_NORESTART;  
}

review source code, 发现在open时候有去init\_work, 这就比较奇怪了, 后来确认, 在open这个devier的时候并没有去init\_work, 提供的  
code是之后修改的. 使用修改后的leds\_strobe.c 文件, 测试没有发生重启情况。

**根本原因：**

workTimeOut没有init,就执行queuework导致KE。

**解决方法：**

在第一次constant\_flashlight\_open的时候对workTimeOut进行初始化一次, 之后open 不需要:

constant\_flashlight\_open()  
{  
 ......       
 static flag = 0;  
  
 if (!flag) {    
 INIT\_WORK(&workTimeOut, work\_timeOutFunc);   
 flag = 1;   
 }  
 ......  
 }

31、work重复初始化引起KE

**问题背景：**

按唤醒键屏亮，定屏，虚拟键无作用，40s后自动重启。

**分析过程：**

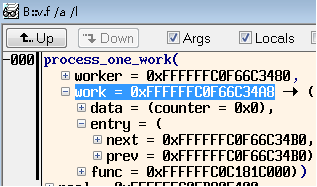
用GAT解开db，并结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），利用工具（E-Consulter.jar）分析（也可以参考FAQ13941），解析出来的调用栈如下：

|  |
| --- |
| == 异常报告v1.4(仅供参考) ==  详细描述: 从错误的地址(0x0000000000000008)读数据，请结合崩溃进程调用栈检查相关代码 异常时间: 3003.395985秒, Tue Jan 26 10:15:56 CST 2016  == CPU信息 == 崩溃CPU信息: CPU5: 进程名: kworker/u16:6, 进程标识符(pid): 8034, 中断: 关 本地调用栈: vmlinux process\_one\_work(worker=0xFFFFFFC0F66C3480, work=0xFFFFFFC0F66C34A8) + 64 <kernel/workqueue.c:2104> vmlinux process\_scheduled\_works() + 8 <kernel/workqueue.c:2244> vmlinux worker\_thread(\_\_worker=0xFFFFFFC0F66C3480) + 588 <kernel/workqueue.c:2311> vmlinux kthread(\_create=0xFFFFFFC02F5ABC90) + 176 <kernel/kthread.c:200> == 栈结束 == 对应汇编指令: 行号 地址 指令 提示 include/asm-generic/atomic-long.h 31 : FFFFFFC0000B8E8C: LDR X21, [X1] ; process\_one\_work()参数 2可能有问题 kernel/workqueue.c 623 : FFFFFFC0000B8E98: AND X21, X21, #-0x100 FFFFFFC0000B8EA4: CSELNE X21, X21 2104: FFFFFFC0000B8EA8: LDR X1, [X21, #0x8] ; 进程停止在这里 当时的寄存器值: X0: 0000000000000081, X1: 0000000000000000, X2: 0000000000000000, X3: FFFFFFC0FB9DA360 X4: 0000000000000001, X5: 000000000000FFF7, X6: FFFFFFC001067007, X7: 65732031206E6168 X8: 3A656D6974202C63, X9: 6F773A656D616E5B, X10: 2665756575716B72, X11: 6E7261772051575D X12: 726F772021676E69, X13: 5F6574616C28206B, X14: 202C656D75736572, X15: 0000000000000000 X16: FFFFFFC0001EA48C, X17: 0000007F9E237BB4, X18: 0000007F9E2ED830, X19: FFFFFFC0F66C34A8 X20: FFFFFFC0F66C3480, X21: 0000000000000000, X22: FFFFFFC0FB80F400, X23: FFFFFFC0C4428000 X24: 00000000FFFFFEF7, X25: FFFFFFC000FCF1FA, X26: FFFFFFC000C67B68, X27: 0000000000000001 X28: 0000000000000089, X29: FFFFFFC0C442BD50, X30: FFFFFFC0000B9528, SP: FFFFFFC0C442BD50 PC: FFFFFFC0000B8EA8 |

上面的报告已经告诉你process\_one\_work()参数 2可能有问题。我们再结合源代码看看是否这样：

static void process\_one\_work(struct worker \*worker, struct work\_struct \*work)  
\_\_releases(&pool->lock)  
\_\_acquires(&pool->lock)  
{  
 struct pool\_workqueue \*pwq = get\_work\_pwq(work);  
 struct worker\_pool \*pool = worker->pool;  
 bool cpu\_intensive = pwq->wq->flags & WQ\_CPU\_INTENSIVE; /\* 这里发生的KE \*/  
  
 ......  
}

从work获取pwq，我们看下work结构体，可以用gdb或trace32查看这个结构体：



pwd从work->data过来，data为NULL，那么pwq也为NULL，那么pwq->wq就出问题了。

这个就印证了分析报告说的work有问题了。

process\_on\_work()是内核原生的代码，属于workqueue，不会有逻辑问题，问题应该出在work上，而且work->func指向的地址也不是一个函数。从正面无法找到问题点，log也没有明显问题，需要看下workqueue的运行机制。

这里需要大家去了解下workqueue的知识，work是使用者自己定义的，然后通过queue\_work()交由linux完成work的工作，所以问题很可能是使用者没有正确使用workqueue。

前面有案例分析《timer重复初始化引起KE》的文章，这题也很有可能，疑惑是work没有初始化就queue\_work()详见《work没有初始化就使用引起KE》

于是仔细排查所有使用work的驱动，结果发现有一处很可能引起重复初始化的代码：

int FL\_Init(void)   
{   
   
 //下面这句会导致workqueue异常   
 INIT\_WORK(&workTimeOut, work\_timeOutFunc);   
   
 printk(" FL\_Init line=%d\n",\_\_LINE\_\_);   
 return 0;   
}  
  
.......  
int FL\_SetTorch(kal\_uint32 state)   
{  
 PK\_DBG("[FL\_SetTorch] !!!\n");   
   
 light\_state = low\_light;   
 if (0 == strobe\_Res)   
 {   
 FL\_Init();   
 }  
........

FL\_Init()可能被重复调用到，那么workTimeOut可能被重复初始化，如果之前workTimeOut已被queue\_work()，那么workTimeOut结构体会被内核链表链接进去。如果还在链表时又对workTimeOut初始化，那么必然把这个内核链表破坏了，进而影响到workqueue的其他work，最后调用work工作时，发生KE了。

代码修改后如下：

int FL\_Init(void)   
{   
   
 //下面这句会导致workqueue异常   
 //INIT\_WORK(&workTimeOut, work\_timeOutFunc);   
   
 printk(" FL\_Init line=%d\n",\_\_LINE\_\_);   
 return 0;   
}  
  
.......  
int FL\_SetTorch(kal\_uint32 state)   
{   
static int flag\_init;   
 PK\_DBG("[FL\_SetTorch] !!!\n");   
   
 light\_state = low\_light;   
 if (0 == strobe\_Res)   
 {   
 FL\_Init();   
 if(flag\_init == 0)   
 INIT\_WORK(&workTimeOut, work\_timeOutFunc);   
 }  
  
........

这样INIT\_WORK()只会调用一次了，就不会出现前面描述的问题。

**根本原因：**

work使用不当。

kernel开发的工程师要特别注意这个问题，work初始化一次就够了。

多注意驱动使用的内核函数的编程规则，避免一些潜在的问题。

**解决方法：**

保证work初始化放在probe()或init()，并且仅执行一次。

32、tasklet没有初始化就是用引起KE

**问题背景：**

擦除坏卡引起KE，必现。

**分析过程：**

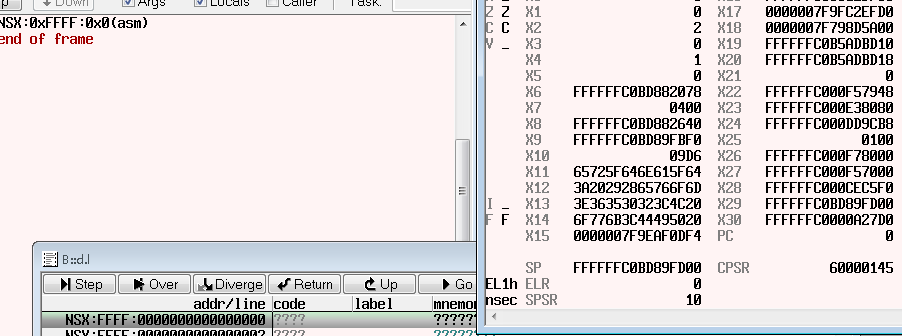
用GAT解开db，并结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），利用工具（E-Consulter.jar）分析（也可以参考FAQ13941），解析出来的调用栈如下：

|  |
| --- |
| == 异常报告v1.4(仅供参考) == 详细描述: tasklet\_hi\_action()调用了错误的函数指针使程序跑到非法地址(0x0000000000000000)执行，请结合崩溃进程调用栈检查相关代码 异常时间: 2799.826556秒, Mon Feb 29 14:21:44 CST 2016  == CPU信息 == 崩溃CPU信息: CPU0: 进程名: ksoftirqd/0, 进程标识符(pid): 3 本地调用栈: ...... ...... vmlinux tasklet\_hi\_action(a=softirq\_vec) + 156 <kernel/softirq.c:556> vmlinux \_\_do\_softirq() + 200 <kernel/softirq.c:279> vmlinux run\_ksoftirqd(cpu=0) + 60 <kernel/softirq.c:677> vmlinux smpboot\_thread\_fn(data=0xFFFFFFC0BD848E40) + 508 <kernel/smpboot.c:160> vmlinux kthread(\_create=0xFFFFFFC0BD848E80) + 212 <kernel/kthread.c:207> == 栈结束 == |

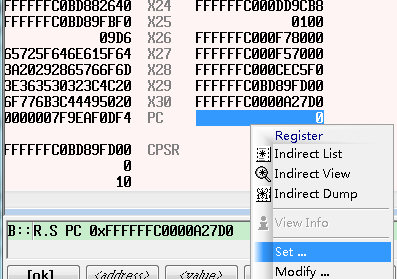
    报告告诉你跑飞了，查看对应的代码：

static void tasklet\_hi\_action(struct softirq\_action \*a)  
{  
 struct tasklet\_struct \*list;  
  
 local\_irq\_disable();  
 list = \_\_this\_cpu\_read(tasklet\_hi\_vec.head);  
 \_\_this\_cpu\_write(tasklet\_hi\_vec.head, NULL);  
 \_\_this\_cpu\_write(tasklet\_hi\_vec.tail, this\_cpu\_ptr(&tasklet\_hi\_vec.head));  
 local\_irq\_enable();  
   
 while (list) {  
 struct tasklet\_struct \*t = list;  
  
 list = list->next;  
  
 if (tasklet\_trylock(t)) {  
 if (!atomic\_read(&t->count)) {  
 if (!test\_and\_clear\_bit(TASKLET\_STATE\_SCHED, &t->state))  
 BUG();  
 t->func(t->data); /\* 这里发生KE \*/

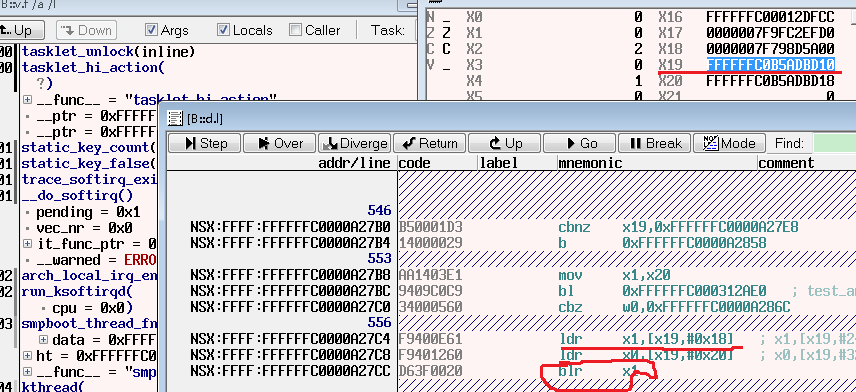
    这时需要启动trace32了，E-Consulter.jar会自动生成debug.cmm，用trace32加载，完成后如图：



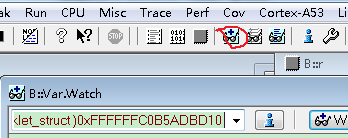
    因为跑飞了，所以什么都看不到，我们要把LR的值赋值给PC（函数调用都会把返回地址赋值给LR）：

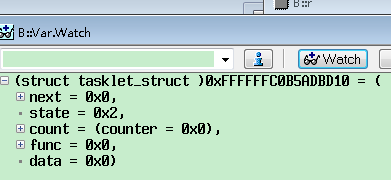


    设定完成后，现场就出来了：



    可以看到BLR X1指令是函数调用，X1的值为0，所以跑飞了，结合代码来看，X1 = t->func，而X19就是t了，我们可以借助trace32查看t：





    明显看到func为NULL引起的KE，那这是哪个tasklet呢？这个结构体无法告诉我们是谁插入tasklet的。

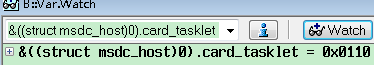
    回头看下这个问题，是擦除坏卡引起，是否和msdc1有关呢，查看log：

|  |
| --- |
| [ 2799.826167] <0> (0)[8203:kworker/0:0][sd]msdc1 -> abort timeout. Card stuck in 7 state, bad card! remove it! <- msdc\_check\_write\_timeout() : L<8280> PID<kworker/0:0><0x200b> [ 2799.826248] <0> (0)[8203:kworker/0:0][sd]msdc1 -> remove the bad card, block\_bad\_card=1,card\_inserted=0 <- msdc\_set\_bad\_card\_and\_remove() : L<2056> PID<kworker/0:0><0x200b> |

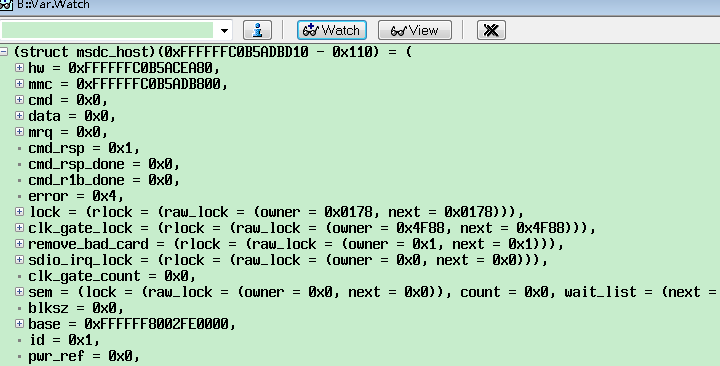
    很可能是msdc的问题，搜索所有调用\_\_tasklet\_hi\_schedule()/\_\_tasklet\_hi\_schedule\_first()的函数，发现有一处和msdc有关，代码如下：

static void msdc\_set\_bad\_card\_and\_remove(struct msdc\_host \*host)  
{  
 ......  
 if (!(host->mmc->caps&MMC\_CAP\_NONREMOVABLE) && (got\_polarity^host->hw->cd\_level))  
 tasklet\_hi\_schedule(&host->card\_tasklet); /\* 这里插入tasklet \*/  
 ......  
}

    是否就是这个host->card\_tasklet的问题呢？我们可以反推下host结构体，然后看后是否是一个正常的mmc\_host结构体，先计算下host到card\_tasklet成员的偏移量：



    结果似乎0x110，然后t - 0x110的地址为msdc\_host结构体地址：



    查看这个结构体的几个关键栏位，确定是正常的msdc host结构体，也就是说正是msdc\_set\_bad\_card\_and\_remove()函数插入的tasklet引起了KE。

    检查下所有用到card\_tasklet的代码，发现就是没有任何代码调用tasklet\_init()来初始化card\_tasklet成员，问题就处在这里。

**根本原因：**

msdc没有初始化card\_tasklet就插入tasklet。

**解决方法：**

不需要tasklet，相关代码需要删除。

33、通话中遮住psensor KE

**问题背景：**

通话中遮住psensor重启。

**分析过程：**

用GAT解开db，并结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），利用工具分析（也可以参考FAQ13941），解析出来的调用栈如下：

|  |
| --- |
| 详细描述: 从错误的地址(0x60C050F9, 不在内核代码空间内)执行程序，请结合崩溃进程调用栈(查看是否有函数指针错误导致)检查相关代码 异常时间: 1586.547690秒   == CPU信息 == 崩溃CPU信息:    CPU0: 进程名: kworker/u:0, 进程标识符(pid): 4043    内核调用栈:     ......   0x60C050F9(参数1:0xC09E7269, 参数2:0x200F0093, 参数3:0x6F5000, 参数4:0x60C050F9) + 0     vmlinux  early\_suspend() + 328     vmlinux  process\_one\_work() + 308     vmlinux  worker\_thread() + 312     vmlinux  kthread() + 140     == 栈结束 == |

调查发现：

* 多台机器出现这样的问题，有些机器容易复现，有些不容易复现。

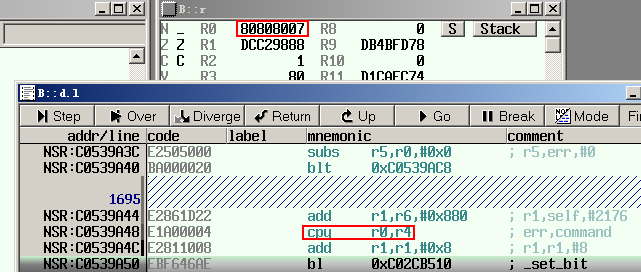
收到反馈，psensor驱动拿掉suspend代码后无复现了，那问题应该是SW引起。  
为了保证代码区不被踩坏，开启MMU protect（FAQ14614），请客户再次复现，拿到log之后，分析如下：

|  |
| --- |
| 详细描述: 从错误的地址(0xECD2A888, 不在内核空间内)读数据，请结合崩溃进程调用栈检查相关代码 异常时间: 251.067248秒   == CPU信息 == 崩溃CPU信息:    CPU1: 进程名: PowerManagerSer, 进程标识符(pid): 724    内核调用栈:     vmlinux    \_set\_bit(参数4:4) + 28     vmlinux    apm\_12d23\_ps\_operate() + 272     vmlinux    hwmsen\_enable() + 392     vmlinux    hwmsen\_unlocked\_ioctl() + 524     vmlinux    do\_vfs\_ioctl() + 144     vmlinux    sys\_ioctl() + 120     == 栈结束 == |

查看对应代码（apm\_12d23.c）：

typedef enum {  
    APM\_BIT\_ALS    = 1,  
    APM\_BIT\_PS     = 2,  
} APM\_BIT;  
   
int apm\_12d23\_ps\_operate(void\* self, uint32\_t command, void\* buff\_in, int size\_in, void\* buff\_out, int size\_out, int\* actualout)  
{  
    ......  
    switch (command)  
    {  
    case SENSOR\_ENABLE:  
        if((buff\_in == NULL) || (size\_in < sizeof(int))) {  
            APM\_ERR("Enable sensor parameter error!\n");  
            err = -EINVAL;  
        } else {  
            value = \*(int \*)buff\_in;  
            if (value) {  
                if ((err = apm\_12d23\_enable\_ps(obj->client, 1)) < 0) {  
                    APM\_ERR("enable ps fail: %d\n", err);  
                    return -1;  
                }  
                set\_bit(APM\_BIT\_PS, &obj->enable);  
......

trace32查看异常原因：



可以看到参数1原本为2，结果传进去的参数变为0x80808007，这个直接导致KE了。  
当时怀疑编译器编译异常，应该直接是mov R1, #2。为何是CPY R1, R4（其中R4为command）？？？百思不得其解，以为是有其他地方将APM\_BIT\_PS定义为：

 #define APM\_BIT\_PS command

修改APM\_BIT\_PS为宏定义：

 #define APM\_BIT\_PS\_VALUE 2

可是编译的结果还是CPY R1, R4。  
后来查看代码才发现SENSOR\_ENABLE也是为2，导致编译直接拿R4优化了。将APM\_BIT\_PS修改为3，编译后验证OK，但是问题还是存在！  
拿基本上定位到R4原本为2，之后被踩坏为0x80808007了，这个一般都是函数调用将R4压栈，踩坏stack后，出栈后就变了，所以怀疑到apm\_12d23\_enable\_ps函数。  
查看函数apm\_12d23\_enable\_ps()（apm\_12d23.c）：

static int apm\_12d23\_enable\_ps(struct i2c\_client \*client, int enable)  
{  
    ......  
    //call calibration  
    if(enable)  
    {  
        apm\_12d23\_fast\_cali(client);  
    }  
    ......  
    return ret;  
}  
   
static void apm\_12d23\_fast\_cali(struct i2c\_client \*client)  
{  
    int i, ret = 0;  
    u8 prox\_data[5] = {0};  
           
    ......  
    quiksort(prox\_data,0,4);  
    ......  
}  
   
void quiksort(u16 a[],int low,int high)  
{  
}

问题就出在**quiksort的参数a是u16的，而apm\_12d23\_fast\_cali传给quiksort的参数prox\_data是u8，直接越界溢出了，刚好踩坏R4！**

**根本原因：**

传给quiksort的参数类型不一致导致越界溢出。

**解决方法：**

修正psensor驱动！

这种问题编译器一定会报警告，优秀的代码，是没有任何警告的。

如果这题将所有警告都修复，那么也不会存在这样的问题了！

34、use after free引起KE

**问题背景：**

待机状态下，按Power键或者自动进入休眠，必现KE。

**分析过程：**

取出mtklog看到有db产生，确实发生了KE（kernel exception），取出db和vmlinux (必须是和当前软件是同一次编译的)后，使用GAT工具解开db，取出SYS\_MINI\_RDUMP，使用 gdb调试：

|  |
| --- |
| $ arm-linux-androideabi-gdb vmlinux SYS\_MINI\_RDUMP GNU gdb (GDB) 7.6 Copyright (C) 2013 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html> This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law. Type "show copying" and "show warranty" for details. This GDB was configured as "--host=x86\_64-linux-gnu --target=arm-linux-android". For bug reporting instructions, please see: <http://source.android.com/source/report-bugs.html>... Reading symbols from KE/vmlinux...done. [New LWP 103] [New LWP 1] [New LWP 2] [New LWP 3] [New LWP 4] [New LWP 5] [New LWP 6] [New LWP 7] [New LWP 8] Core was generated by `console=tty0 console=ttyMT0,921600n1 root=/dev/ram vmalloc=496M slub\_max\_order='. #0 0xd9027400 in ?? () (gdb) bt #0 0xd9027400 in ?? () #1 0xc00985c0 in early\_suspend (work=<optimized out>) at kernel-3.10/kernel/power/earlysuspend.c:144 #2 0xc0042b60 in process\_one\_work (worker=0xdaf6d000, work=0xc0e056a0 <early\_suspend\_work>) at kernel-3.10/kernel/workqueue.c:2216 #3 0xc0043098 in worker\_thread (\_\_worker=0xd90273dc) at kernel-3.10/kernel/workqueue.c:2348 #4 0xc004c0bc in kthread (\_create=0xdbc69df0) at kerne l-3.10/kernel/kthread.c:200 #5 0xc000f308 in ret\_from\_fork () at kernel-3.10/arch/ arm/kernel/entry-common.S:91 #6 0xc000f308 in ret\_from\_fork () at kernel-3.10/arch/ arm/kernel/entry-common.S:91 Backtrace stopped: previous frame identical to this frame (corrupt stack?) (gdb) f 1 #1 0xc00985c0 in early\_suspend (work=<optimized out>) at kernel-3.10/kernel/power/earlysuspend.c:144 144 pos->suspend(pos); (gdb) list 139 if (pos->suspend != NULL) { 140 if (!(forbid\_id & (0x1 << count))) { 141 /\* if (earlysuspend\_debug\_mask & DEBUG\_VERBOSE) \*/ 142 pr\_warn("ES handlers %d: [%pf], level: %d\n", count, pos->suspend, 143 pos->level); 144 pos->suspend(pos); 145 } 146 count++; 147 } 148 } |

一套命令下来就知道问题发生在发现是死在了 kernel-3.10/kernel/power/earlysuspend.c:144 位置，而且是跑飞（#0的地址0xd9027400不是有效的代码地址），查看具体挂掉的原因，打印汇编代码：

|  |
| --- |
| (gdb)disas  0xc009859c <+348>: b 0xc00984e8 <early\_suspend+168>  0xc00985a0 <+352>: ldr r3, [r4, #8] 0xc00985a4 <+356>: mov r1, r6 0xc00985a8 <+360>: movw r0, #15956 ; 0x3e54 0xc00985ac <+364>: movt r0, #49338 ; 0xc0ba 0xc00985b0 <+368>: bl 0xc09849d8 <printk> 0xc00985b4 <+372>: ldr r3, [r4, #12] 0xc00985b8 <+376>: mov r0, r4 **0xc00985bc <+380>: blx r3** => 0xc00985c0 <+384>: b 0xc0098550 <early\_suspend+272> 0xc00985c4 <+388>: addsgt r3, r10, r8, lsr sp 0xc00985c8 <+392>: rscgt r5, r0, r8, asr #12 End of assembler dump. (gdb) info reg r0 0xd90273dc 3640816604 r1 0x60785a 6322266 r2 0x0 0 r3 0xd90273e8 3640816616 r4 0xd90273dc 3640816604 r5 0xc0f08180 3236987264 r6 0xc 12 r7 0xc0e05648 3235927624 r8 0xdbc54400 3687138304 r9 0xdafcc030 3673997360 r10 0xdb70b6d8 3681597144 r11 0xdafcddec 3674004972 r12 0xc10af654 3238721108 sp 0xdafcddd0 0xdafcddd0 lr 0xc00985c0 3221849536 pc 0xc00985c0 0xc00985c0 <early\_suspend+384> cpsr 0x200b0013 537591827 |

可以很明确看到=>之上的汇编指令（根据AAPCS，非#0帧的地址都是返回地址，是还没执行过的指令，而前一条汇编指令则是函数跳转指令）BLX正是函数跳转指令。肯定是pos->suspend错误了。那这个suspend是哪个驱动通过register\_early\_suspend()注册的呢？现有db里的信息无法知道，需要添加log才行。

先看下pos在内存中的位置：

|  |
| --- |
| (gdb) p pos  $2 = (struct early\_suspend \*) **0xd90273dc** |

现在要做的就是要想办法把这个是哪里来得给抓出来，分析earlysuspend.c 内核代码 early\_suspend 函数，pos是从early\_supend\_handers这个链表得来的，所以看下early\_supend\_handers是在哪里加载的，追踪代码会发现追踪是在 register\_early\_suspend 函数注册的，所以应该只要在这个函数加打印，再抓开机log就可以把错误的pos->suspend 揪出来！打印信息如下：

void register\_early\_suspend(struct early\_suspend \*handler)  
{  
    struct list\_head \*pos;  
    mutex\_lock(&early\_suspend\_lock);  
    list\_for\_each(pos, &early\_suspend\_handlers) {  
        struct early\_suspend \*e;  
        e = list\_entry(pos, struct early\_suspend, link);  
        if (e->level > handler->level)  
            break;  
    }  
    printk("#---^\_^-->>>[%s],[0x%lx],[0x%lx],[%pf]\n", \_\_func\_\_, (long)handler, (long)&handler->suspend, handler->suspend);  
    list\_add\_tail(&handler->link, pos);  
    early\_suspend\_count++;  
    if ((state & SUSPENDED) && handler->suspend)  
        handler->suspend(handler);  
    mutex\_unlock(&early\_suspend\_lock);  
}

编译、烧机，复现问题（因为有加打印代码改动，所以有可能需要重新复现问题抓db再使用gdb调试打印 pos->suspend 的值于开机log做对比），抓开机log搜索：0xd90273e8：

|  |
| --- |
| [ 21.423978]<2>.(1)[1:swapper/0]#---^\_^-->>>[register\_early\_suspend],[0xd90273dc],[0xd90273e8],[**gfx1xm\_early\_suspend**] |

现在就很明显了，凶手就是 gfx1xm\_early\_suspend()函数，KE问题锁定在gfx1xm驱动范围内！  
查看gfx1xm驱动代码发现xxx\_probe()里面有register\_early\_suspend()函数：

gfx1xm\_dev->early\_fp.level = EARLY\_SUSPEND\_LEVEL\_DISABLE\_FB - 1,  
gfx1xm\_dev->early\_fp.suspend = gfx1xm\_early\_suspend,  
gfx1xm\_dev->early\_fp.resume = gfx1xm\_late\_resume,  
register\_early\_suspend(&gfx1xm\_dev->early\_fp);

而复现问题的情况是，**当gfx1xm设备不存在的时候才会出现，而在设备remove后会释放掉gfx1xm\_dev结构体，导致了注册的suspend的结构体内存也变为空闲，很可能被其他模块申请去使用了，这个就是典型的use after free的情况。**

正确的解法是：设备不存在的时候应该remove掉这个reigster，而当前的驱动xxx\_remove()里面却没有对应的反注册函数导致了此问题！

这样一分析，那么解决问题就简单了，直接在退出函数加上反注册函数就行了：

static int gfx1xm\_remove(struct spi\_device \*spi)  
{  
 struct gfx1xm\_dev \*gfx1xm\_dev = spi\_get\_drvdata(spi);  
 FUNC\_ENTRY();  
 klog("unregister\_early\_suspend(&gfx1xm\_dev->early\_fp)\n");  
 unregister\_early\_suspend(&gfx1xm\_dev->early\_fp);  
 ......  
}

编译、烧机，开机 验证ok。

**根本原因：**

use after free。

驱动suspend结构体在被remove掉后被释放，但由于没有反注册suspend，导致early suspend过程中使用了这个结构体引起KE。

**解决方法：**

在remove()里添加反注册suspend：unregister\_early\_suspend(&gfx1xm\_dev->early\_fp);

35、内存泄露引起KE

**问题背景：**

用户反馈系统重启。

**分析过程：**

用GAT解开db，查看\_\_exp\_main.txt：

|  |
| --- |
| Exception Class: Kernel (KE) **PC is at [<ffffffc0003e2800>] mrdump\_mini\_ke\_cpu\_regs**+0x50/0xc4 |

看到这个信息，表示系统主动调用kernel panic。这时需要看SYS\_KERNEL\_LOG，搜索关键字Kernel panic - not syncing：

|  |
| --- |
| [35649.303767]<3>-(3)[28285:netd]Kernel panic - not syncing: **Out of memory and no killable processes...** |

发现panic的原因是OOM，这行log是在这里的代码打印的：

[kernel-3.18](http://10.16.20.15/alps-mp-m0-mp9/xref/kernel-3.18/)/[mm](http://10.16.20.15/alps-mp-m0-mp9/xref/kernel-3.18/mm/)/[oom\_kill.c](http://10.16.20.15/alps-mp-m0-mp9/xref/kernel-3.18/mm/oom_kill.c):

void out\_of\_memory(struct zonelist \*zonelist, gfp\_t gfp\_mask, int order, nodemask\_t \*nodemask, bool force\_kill)  
{  
 const nodemask\_t \*mpol\_mask;  
 struct task\_struct \*p;  
 unsigned long totalpages;  
 ......  
 p = select\_bad\_process(&points, totalpages, mpol\_mask, force\_kill);  
 /\* Found nothing?!?! Either we hang forever, or we panic. \*/  
 if (!p) {  
 dump\_header(NULL, gfp\_mask, order, NULL, mpol\_mask);  
 panic("Out of memory and no killable processes...\n");  
 }  
 ......  
}

 OOM表示kernel已没有内存可以回收了，需要看哪里泄漏了。可能的原因有：

1. 用户进程泄漏，LMK不是所有进程都去回收的，是根据adj来判断，低于0不回收，由init启动的service都不回收，像system\_server，surfaceflinger等是不会被回收的，而如果这些进程发生内存泄漏，最后只能OOM了。
2. kernel泄漏，比如slub、vmalloc泄漏。

一般OOM前会印系统内存信息，有助于我们判断哪里泄漏，在panic前有看到如下log：

|  |
| --- |
| [35649.261853]<3> (3)[28285:netd]**netd invoked oom-killer: gfp\_mask=0x3000d0, order=2, oom\_score\_adj=-1000** [35649.261873]<3> (3)[28285:netd]CPU: 3 PID: 28285 Comm: netd Tainted: G W 3.10.65+ #1 [35649.261883]<3> (3)[28285:netd]Call trace: [35649.261901]<3> (3)[28285:netd][<ffffffc000088f50>] dump\_backtrace+0x0/0x16c [35649.261915]<3> (3)[28285:netd][<ffffffc0000890cc>] show\_stack+0x10/0x1c [35649.261931]<3> (3)[28285:netd][<ffffffc0009a03b8>] dump\_stack+0x1c/0x28 [35649.261947]<3> (3)[28285:netd][<ffffffc000150244>] dump\_header.isra.14+0x6c/0x1b0 [35649.261960]<3> (3)[28285:netd][<ffffffc000150ef8>] out\_of\_memory+0x2ec/0x2fc [35649.261975]<3> (3)[28285:netd][<ffffffc000155b54>] \_\_alloc\_pages\_nodemask+0x788/0x79c [35649.261991]<3> (3)[28285:netd][<ffffffc0000984a4>] copy\_process+0x114/0x10fc [35649.262003]<3> (3)[28285:netd][<ffffffc00009953c>] do\_fork+0x7c/0x3c4 [35649.262015]<3> (3)[28285:netd][<ffffffc00009991c>] SyS\_clone+0x10/0x1c [35649.262023]<3> (3)[28285:netd]Mem-Info: ...... [35649.262096]<3> (3)[28285:netd]active\_anon:552580 inactive\_anon:138202 [35649.262096]<3> active\_file:1172 inactive\_file:1532 isolated\_file:32 [35649.262096]<3> unevictable:631 dirty:23 writeback:1590 [35649.262096]<3> free:1971 slab\_reclaimable:2893 slab\_unreclaimable:7057 [35649.262096]<3> mapped:1387 shmem:123 pagetables:2379 [35649.262139]<3> (3)[28285:netd]DMA free:7884kB min:6932kB**low:19372kB** high:21104kB **active\_anon:2210320kB inactive\_anon:552808kB** active\_file:4688kB inactive\_file:6128kB unevictable:2524kB isolated(file):128kB **present:3072960kB** managed:3004616kB dirty:92kB writeback:6360kB mapped:5548kB shmem:492kB slab\_reclaimable:11572kB slab\_unreclaimable:28228kB kernel\_stack:6160kB pagetables:9516kB pages\_scanned:16804 all\_unreclaimable? yes [35649.212755]<3> (3)[28285:netd]powerkey\_kick:primary\_display\_trigger:2,6 [35649.262154]<3> (3)[28285:netd]lowmem\_reserve[]:[35649.262161]<3> (3)[28285:netd] 0 0 0 [35649.262173]<3> (3)[28285:netd]**DMA**: 1922\*4kB (UEM) 1\*8kB (R) 7\*16kB (R) 1\*32kB (R) 0\*64kB 2\*128kB (R) 0\*256kB 0\*512kB 0\*1024kB 0\*2048kB 0\*4096kB = **8096kB** [35649.262230]<3> (3)[28285:netd]3923 total pagecache pages [35649.262239]<3> (3)[28285:netd]294 pages in swap cache [35649.262249]<3> (3)[28285:netd]Swap cache stats: add 211983, delete 211689, find 245614/274392 [35649.262257]<3> (3)[28285:netd]**Free swap = 0kB** [35649.262265]<3> (3)[28285:netd]Total swap = 524284kB [35649.302860]<3> (3)[28285:netd]768240 pages RAM [35649.302874]<3> (3)[28285:netd]16722 pages reserved [35649.302882]<3> (3)[28285:netd]46561 pages shared [35649.302891]<3> (3)[28285:netd]740566 pages non-shared [35649.302900]<3> (3)[28285:netd][ pid ] uid tgid total\_vm **rss** nr\_ptes swapents oom\_score\_adj name [35649.303030]<3> (3)[28285:netd][ 236] 1000 236 138735 1303 63 862 -1000 surfaceflinger [35649.303251]<3> (3)[28285:netd][ 277] 1021 277 12652 190 26 137 -1000 mtk\_agpsd [35649.303303]<3> (3)[28285:netd][ 279] 1021 279 4003 145 8 61 -1000 mnld [35649.303467]<3> (3)[28285:netd][ 305] 0 305 242916 2214 114 2237 -1000 main [35649.303508]<3> (3)[28285:netd][ 312] 1023 312 4074 1078 9 45 -1000 sdcard [35649.303536]<3> (3)[28285:netd][ 330] 0 330 107 2 4 9 -1000 ku.sud [35649.303549]<3> (3)[28285:netd][ 332] 0 332 112 15 4 1 -1000 ku.sud [35649.303562]<3> (3)[28285:netd][ 424] 0 424 790291 **668810** 1547 121425 -1000 **ku.sud** [35649.303576]<3> (3)[28285:netd][ 609] 2000 609 3731 5 8 158 -1000 emdlogger1 [35649.303590]<3> (3)[28285:netd][ 610] 1001 610 6958 46 13 68 -1000 gsm0710muxd [35649.303604]<3> (3)[28285:netd][ 760] 0 760 2714 29 6 122 -1000 debuggerd\_real [35649.303617]<3> (3)[28285:netd][ 853] 1001 853 12059 132 22 151 -1000 mtkrild [35649.303631]<3> (3)[28285:netd][ 1473] 0 1473 65 3 5 7 -1000 kd [35649.303644]<3> (3)[28285:netd][ 5758] 0 5758 63 1 5 7 -1000 ktools [35649.303658]<3> (3)[28285:netd][10776] 0 10776 65 10 5 0 -1000 kd [35649.303676]<3> (3)[28285:netd][26884] 1010 26884 4356 245 9 0 -1000 wpa\_supplicant [35649.303689]<3> (3)[28285:netd][27143] 1014 27143 2586 79 6 0 -1000 dhcpcd [35649.303703]<3> (3)[28285:netd][27915] 1003 27915 67212 5752 61 0 -1000 bootanimation [35649.303717]<3> (3)[28285:netd][28284] 0 28284 17478 922 34 0 -1000 zygote64 [35649.303731]<3> (3)[28285:netd][28285] 0 28285 3226 194 7 0 -1000 netd [35649.303744]<3> (3)[28285:netd][28286] 0 28286 9923 594 19 0 -1000 mediaserver [35649.303757]<3> (3)[28285:netd][28415] 0 28415 207610 12988 61 0 -1000 main ...... [35649.303767]<3>-(3)[28285:netd]Kernel panic - not syncing: Out of memory and no killable processes... |

我们来看如何解读这个log，第1行log表示netd要申请内存申请不到，需要启动LMK释放一些内存，申请的尺寸是order = 2，2^2 = 4 page = 4 \* 4K。

zone只有一个dma，管理的内存大小是：3G（**present:3072960kB**）。

DMA zone只有8M（**8096kB**）的内存了，已经低于低水位了（**low:19372kB**），存在泄漏，而且zram也用光了（**Free swap = 0kB**）。

我们发现**active\_anon:2210320kB inactive\_anon:552808kB**这2个加起来超过2.7G，表示用户进程泄漏了。

查看用户进程rss栏位，找出最大的那个进程是**ku.sud**，泄漏内存大小是**668810**\* 4K = 2.6G。很明显需要分析这个进程泄漏原因。

用户进程泄漏排查方法，可以参考：[FAQ14715]如何分析native memory leak

**根本原因：**

用户进程泄漏。

**解决方法：**

排查这个用户进程泄漏问题。

PS：kernel内存管理是很大一个系统，需要较多背景知识，建议大家了解下这个背景，对分析问题很有帮助。